Reasoning about Java programs in higher order logic using PVS and Isabelle

Marieke Huisman

ISBN 90-9014440-4
IPA Dissertation Series 2001-03

Typeset with LaTeX 2_ε
Printed by Print Partners Ipskamp, Enschede
Cover design by Arjan Huisman, ja_mus@excite.com

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Reasoning about Java programs in higher order logic using PVS and Isabelle

een wetenschappelijke proeve op het gebied
van de Natuurwetenschappen, Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Katholieke Universiteit Nijmegen,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op
donderdag 1 februari 2001
des namiddags om 3.30 uur precies
door

Marieke Huisman

geboren op 3 mei 1973 te Utrecht

Promotor:

Prof. dr. H.P. Barendregt

Copromotores:

Dr. B.P.F. Jacobs

Dr. ir. H. Meijer

Manuscriptcommissie:

Prof. dr. T. Nipkow	Techische Universität München, Duitsland
Prof. dr. A. Poetzsch-Heffter	FernUniversität Hagen, Duitsland
Prof. dr. S.D. Swierstra	Universiteit Utrecht

Preface

And now, after four years the job is done, and the thesis is printed...

And thus the time has come to thank the people that helped me in all kinds of ways during this period. First of all I wish to thank my supervisors: Bart Jacobs, Hans Meijer and Henk Barendregt. Bart, being my daily supervisor and the leader of the LOOP project, has been closely involved in everything. He gave direction to my research, and always provided useful feedback. Our meetings were always very inspiring and pleasant, no matter whether we discussed research or the latest gossips. Hans always helped me in keeping the overall view of what I was doing and carefully read everything I wrote, finding many mistakes that I simply would have overlooked. Henk's questions were always inspiring and often helped me in improving my explanations.

The work that is presented in this thesis has been done in the context of the LOOP project. I enjoyed the collaboration with the other team members: Joachim van den Berg, Ulrich Hensel, Erik Poll and Hendrik Tews. Within the project, there was a good and open atmosphere, with room and time for discussions, collaboration, and fun. Most of the work presented in this thesis is co-authored with these people. I would like to thank all of them for the pleasant team-spirit. I also would like to thank David Griffioen, who is also one of my co-authors, and who taught me how to be critical about my own work.

Also, I thank everybody who read (parts of) this thesis. Apart from the people mentioned above, these were the members of the reading committee: Tobias Nipkow, Arnd Poetzsch-Heffter, and Doaitse Swierstra. Also Wishnu Prasetya and Kim Sunesen gave useful suggestions from improvements. Many thanks also to everybody that helped me to understand ISABELLE, besides Tobias these were: Larry Paulson, Markus Wenzel, David von Oheimb and Florian Kammüller. Also, I want to thank Mike Gordon and the people from the Automatated Reasoning Group for hosting me in the Computer Lab in Cambridge.

Special thanks to my office mates: D.J. Chen, Franc Grootjen, Peter Lambooij and Judi Romijn. With each of them, I had many interesting conversations, while having a mug of tea.

Further, I would like to thank Frits Vaandrager, as the leader of the ITT group, Hanno Wupper, who was one of my supervisors in my first year, and Mirése Willems, for being a great secretary. Also I want to thank the colleagues and guests from ITT, in particular Mariëlle Stoelinga, Ansgar Fehnker, Angelika Mader, Thomas Hune, Harco Kuppens, Wim Janssen, Ulrich Hannemann, Jozef Hooman, André van den Hoogenhof, Adriaan de Groot, and Theo Schouten, for all the enjoyable lunches, coffee breaks, ice times, and evenings in the pub.

Many thanks also to my brother Arjan, and my *paranimphs* Carolijn and Joachim (again) for helping me in preparing everything for the thesis, the defense and the party afterwards.

Finally, I like to thank my parents, Ria and Bas Huisman, for always supporting me during these four years.

Contents

Chapter 1

Introduction

Already since the beginning of computer science, program correctness is one of the important issues. Ideally, all software should be proven correct, *i.e.* shown to be satisfying its specification. Typical properties that one would like to verify of programs (or procedures) are the following.

- The program terminates under certain conditions.
- The program throws an exception under certain conditions.
- The input and output state of a program are related in a particular way, *i.e.* the program has a certain behaviour.
- A property is invariant, *i.e.* it is true in all visible states of the program.
- The program changes only particular variables (possibly none), the other variables are unchanged. This is a more technical property, which is often needed in the verification of other properties.

To be able to do this, both specification and programming language should have a formal semantics, *i.e.* a semantics that can be described in logic. Only then it is possible to formulate and establish the correctness of a program formally.

To achieve this, research concentrated on describing semantics of certain programming languages and developing formal methods to prove program correctness, *e.g.* Hoare logic, or to calculate correct programs (*e.g.* the weakest precondition calculus). These proof methods describe how the correctness of a program can be established step-by-step. But, in order to get a nice and simple semantics and proof method, the programming languages under consideration are neat and simple; they are mainly toy programming languages. And even for these toy programming languages, proving program correctness is very hard. Already for small programs, the correctness proofs become quite large, since every detail has to be spelled out. Many of the proof steps can be applied mechanically, they do not introduce any new ideas, but just require careful calculations. Usually, there are only a few steps in a proof where creativity is required, the other steps are more or less bookkeeping.

This work has been influential, since it showed that theoretically there is a possibility to establish program correctness, but unfortunately, it did not provide a full solution to the quest for program correctness. The programs that one actually would like to verify are large, and written in real programming languages, with all their messy semantical details. Thus, work on

program verification and formal methods continued, trying to find the right balance between feasibility, ease of use, and soundness of the method. Ideally, it should be possible to verify a program written in an arbitrary programming language (without any restrictions on the parts of the language that can be used), with reasonable effort and within reasonable time. And of course, the verification should be correct (in particular not accept incorrect programs).

This thesis discusses new developments in the field of program verification, which make verification of programs written in a real-world programming language more feasible. The initial impetus for the work in this thesis has been given by several recent developments in computer science.

First of all, a new programming paradigm has become popular, namely that of object-oriented programming. The first object-oriented languages date back to the sixties and seventies (SIMULA [DMN70], SMALLTALK [GR83]), but with the introduction of C++ and JAVA the paradigm has become increasingly popular. In an object-oriented setting, a program consists of a number of objects, interacting with each other. Each object is described by a class, which contains field and method declarations. Classes resemble modules in the sense that they can be reused in different applications. The possibility to reuse classes makes program verification more important (as it is desirable to use a completely verified class) and also more cost efficient: since verifications usually take much time, it is better to verify program code that is used more often.

Typically, object-oriented programming languages come with a library of predefined classes. These classes provide all kind of basic behaviour and are used in many applications. Formal specification and verification of the methods in these library classes can increase the usability of these classes and the reliability of the programs based on them. For example, after a method is verified, it is clear under which conditions the method will throw an exception or what postcondition it will satisfy. Typical for object-oriented languages is the possibility to extend classes (as so-called subclasses) and to redefine methods in subclasses. Which method is actually used, depends on the run-time type of an object, *i.e.* the binding of the methods is done dynamically. Therefore, this is often called dynamic or late binding. It is a challenge to describe dynamic binding formally.

As mentioned above, JAVA is one of the better known object-oriented languages. Initially called OAK, it is loosely based on C++. The language has been stripped down to a bare minimum; as it was intended to work for consumer electronic devices, which often used chips with limited program space. Furthermore, it was designed to allow programmers to more easily support dynamic, changeable hardware [Eng98].

There is no (official) formal semantics of JAVA available (but it is an important research topic at the moment). Since JAVA translates to so-called bytecode, which is platform independent, it is used in many internet applications. This is one of the main reasons why JAVA has become one of the most popular and widely-used programming languages so quickly. Several dialects of JAVA exist, among which there is JAVACARD. This is a subset of JAVA, which is used to program smart cards. Security is an important issue for smart cards, thus for smart cards applications verification is even more important.

Furthermore, developments in formal methods have led to powerful tools which can assist in program verification. These tools can perform many of the trivial steps in verification without user interaction, allowing the user to concentrate on the crucial points. A wide range of different kinds of tools is available for this purpose. In this thesis we focus on the use of interactive proof tools for higher order logic, but other kinds of tools, such as model checkers and automated

(first order) theorem provers, also have shown their use in program verification.

An interactive proof tool is a system which allows a user to build a proof interactively. The user states a goal that has to be proven. The user applies proof commands to this goal, and after each step the theorem prover shows the remaining proof obligations, thus doing all the bureaucratic work involved in proving. Also, as all the calculations and logical inferences are done by the machine, instead of by the user, this prevents the introduction of clerical errors. However, the proof is still constructed by the user, not by the machine. In the last decade, these interactive proof tools have improved significantly, providing more powerful proof commands to the user. Thus, the theorems that can be proven with a single command have become more and more complex.

To make program verification using interactive theorem provers really feasible, the proof tool should be able to do large verifications without much user interference. Ideally, all the bookkeeping steps are done by the machine, the user only has to interfere at the crucial points in the proof, *e.g.* at loop entrance and recursive method calls.

Finally, the last development which is of interest for this thesis is the use of coalgebras to give a semantics to objects. Coalgebras are functions of the form $c\colon X \to F(X)$, where F is a functor and X is called the carrier. Coalgebras are the formal dual of algebras, which are functions of the form $a\colon F(X) \to X$. Algebras are used to construct elements in the carrier set. For example, a group $< G, +_G, -_G, 0_G >$ can be described as an algebra $a\colon (G \times G)+G+1 \to G$, which is composed of the functions $+_G$, $-_G$ and the constant 0_G. (In this type $+$ is the direct sum and 1 the one element set.)

In contrast, coalgebras only allow to make observations and modifications on the elements in the carrier set: their elements cannot be constructed. The standard example of a coalgebra is infinite lists with elements of type A, described by a coalgebra c with type $X \to A \times X$. This coalgebra has the following intended meaning. If $l\colon X$ is an infinite list, then $c\,l = (\mathsf{head}\,l, \mathsf{tail}\,l)$. These functions $\mathsf{head}\colon X \to A$ and $\mathsf{tail}\colon X \to X$ can therefore be defined from the coalgebra c. Using head and tail the i^{th} element of the list can be observed by applying tail $i-1$ times, followed by an application of head. The "whole" list however can never be created. Typically, coalgebras are used to describe possibly infinite behaviour of systems, for which there exists only a notion of behavioural equivalence. For more information on coalgebras see [JR97]. Objects are another typical example that can be described using coalgebras [Rei95]. The state of an object is not visible for the outside world, but it can be observed and modified, using the available methods. A notion of (observationally) equality or bisimilarity exists, which describes when two objects cannot be distinguished by their behaviour.

The way coalgebras are used in this thesis is fairly superficial, it mainly provides the basis for our representation of classes. However, it is important to recognise that the concept of coalgebras is behind the work presented here, because this recognition immediately leads to related concepts, such as invariance, bisimilarity and modal operators (leading to a special Hoare logic, as presented in this thesis). Although it is not necessary to be familiar with the theory of coalgebras to understand the work presented in this thesis, this familiarity can give new insights in possible extensions of this work.

These three developments (interest in semantics of object-oriented programs, development of powerful proof tools, and recognition of the usefulness of coalgebras to describe (object-oriented) semantics) form the basis for the LOOP project. The LOOP project, which is short for Logic of Object Oriented Programming, aims at the specification and verification of object-oriented specifications and programs. For the verifications powerful proof tools are used, in

particular PVS and ISABELLE.

The LOOP project started in 1997 as a joint project between the universities of Nijmegen and Dresden. As mentioned above, the basis of the project is formed by the idea that coalgebras can be used to describe a semantics for objects [Rei95]. Bart Jacobs and Ulrich Hensel developed a set of PVS theories which capture the semantics of so-called class specifications, *i.e.* classes consisting of field and method declarations and assertions, describing the behaviour of its methods. Based on these assertions, properties about the specifications can be proven. Typical properties that are proven about class specifications are class invariants and the existence of a refinement relation between specifications. For each class new PVS theories have to be constructed, but this can be done according to a standard pattern. Therefore, work started on programming a compiler that automatically translates class specifications into PVS theories. To write down class specifications, a language called CCSL, for Coalgebraic Class Specification Language, was developed. Initially, the assertions describing the specification were written in the PVS specification language; later also a special assertion language for CCSL has been developed [RJT00].

From 1998 on, the LOOP project broadened itself and also paid attention to JAVA. The basic semantics of JAVA statements and expressions was described in PVS and the LOOP compiler was adapted so that it could also translate JAVA classes into PVS theories. Later, during 1999, the LOOP compiler was extended so that it also could generate ISABELLE theories. Our verification of JAVA classes heavily rely on automatic rewriting, and we wanted to investigate whether the powerful rewriting strategies of ISABELLE would be useful to reason about JAVA programs. At the moment, the LOOP compiler translates almost all of sequential JAVA into either PVS or ISABELLE. The LOOP compiler has been applied to several larger case studies (see Chapter 7 and [PBJ00]). Also, it has been applied to a substantial subset of 100 small, but tricky JAVA programs, constructed by Jan Bergstra [BL99]. These programs, which are used in a course on *empirical semantics of* JAVA, describe different, non-trivial aspects of the JAVA semantics. They form a very good independent benchmark to test our formalisation of the JAVA semantics.

Initially the user statements, *i.e.* the properties that one wishes to prove about the JAVA program at hand, had to be given in the input language of the theorem prover. Current work in the LOOP project focuses on an annotation language for JAVA, called JML. JML allows the user to write assertions about the program in the program code itself. Currently, the LOOP compiler is extended so that it also analyses the program annotations and generates appropriate proof obligations for these annotations. In this thesis, the language JML is already used to denote method and class specifications, but the translation from these annotations to proof obligations in PVS or ISABELLE is still done manually. Thus, this thesis is on the border between two different phases in the project: the JAVA semantics is already established and incorporated in the LOOP compiler, but the JML semantics is still under investigation and not incorporated in the compiler.

This thesis describes the following aspects of the JAVA branch of the LOOP project.

- **The Java semantics.** For the notion of classes a translation is discussed, which can translate the program code into a mathematical description of the class (based on coalgebras).

- **Tool support.** Within the project, a compiler is built, which translates JAVA classes into theories that can be used as input for the theorem prover PVS and ISABELLE. Actual

reasoning about JAVA classes is done within these theorem provers. The use of these two proof tools is discussed in detail in this thesis.

- **Reasoning about Java.** To facilitate proving properties of JAVA classes, proof methods tailored to JAVA are developed, *e.g.* based on traditional Hoare logic. The purpose of these proof methods is to make verification of JAVA classes more efficient. This thesis gives ample attention to these proof rules.

The thesis starts by describing the basic ingredients of the project in the first chapters. This introduction concludes by giving a short overview of typical terminology for object-orientation. Chapter 2 describes the JAVA semantics underlying the project. This semantics is given in a simple type theory, which can easily be translated into an input language for a higher order theorem prover. Chapter 3 introduces interactive theorem proving in more detail, describes the proof tools PVS and ISABELLE and gives a general, but detailed comparison of their features and capabilities. Then, Chapter 4 describes the LOOP tool and discusses some simple verifications.

Subsequently, this thesis discusses the actual verification of JAVA programs. To make verification more feasible, special proof techniques are required. Chapter 5 looks at verifications within a single class. A Hoare logic is introduced, which is tailored towards reasoning about JAVA programs. Chapter 6 then describes a more structured way to describe specifications (both of methods and classes). It introduces the language JML, which allows a programmer to write specifications in his/her JAVA program. From these annotations, appropriate proof obligations can be generated.

The last chapter, Chapter 7 describes two larger case studies that have been done within the project. The first one concerns a verification in PVS of a class invariant of the class `Vector` from the standard JAVA library. The second case study deals with the hierarchy of collection classes. It verifies an (abstract) implementation of a collection class, using specifications of abstract methods and methods from other classes, *i.e.* the verification is done in a modular way. This verification is done in ISABELLE. Finally, Chapter 8 gives conclusions, and also discusses and compares experiences with PVS and ISABELLE in the two case studies.

Much of the work described in this thesis is joint work with (some of) the other (former) members of the LOOP project: Joachim van den Berg, Martijn van Berkum, Ulrich Hensel, Bart Jacobs, Erik Poll, and Hendrik Tews. Much of the work reported on here also has been published elsewhere. The first paper that reported on the JAVA branch of the LOOP project [JBH+98] gives a general overview of the project. After that, several papers have been published which described one or two aspects of the project in more detail. Below, for each chapter it is discussed who contributed what, and where it has been published.

Chapter 2: The JAVA semantics, as it is discussed in this chapter is developed by Bart Jacobs, with significant improvements (based on verification experiences) suggested by Joachim van den Berg, Erik Poll, and the author. Several papers have appeared, presenting part of this JAVA semantics. In [HJ00b] the semantics of the statements and expressions (as explained in the Sections 2.2, 2.3, 2.4) is discussed. The explanation of the memory model, as described in Section 2.5 is based on [BHJP00]. The semantics of classes (Section 2.6) appeared as [HJ00a].

Chapter 3: The comparison of PVS and ISABELLE/HOL presented in this chapter is based on joint work with David Griffioen [GH98].

Chapter 4: Most of the work reported on in Chapter 4 has not been published elsewhere. The LOOP compiler is mainly implemented by Joachim van den Berg, Martijn van Berkum, Ulrich Hensel, Bart Jacobs and Hendrik Tews. The extension to ISABELLE has been programmed by the author. Two of the example verifications have been published in [HJ00a].

Chapter 5: The Hoare logic presented in this chapter is developed by the author, with improvements based on suggestions by Joachim van den Berg and Bart Jacobs. This logic has been presented in [HJ00b].

Chapter 6: The language JML is developed by the group of Gary Leavens at Iowa State University [LBR98]. The semantics, on which the proof obligations are based, is still under development. This work is done by Joachim van den Berg, with contributions by Bart Jacobs and Erik Poll.

Chapter 7: The first case study described in this chapter is joint work with Bart Jacobs and Joachim van den Berg. It has been reported on in [HJB00]. The second case study is done by the author, with suggestions about the specifications by Erik Poll. It has not been published elsewhere.

1.1 Basic terminology of object-orientation

Even though object-orientation is popular at the moment, and one of the big buzz words, there is still a lot of confusion about many of the terms used to describe the various concepts. This section does not try to give a full introduction into object-orientation, but it tries to fix the terminology, much like for JAVA, which is used in the rest of the thesis.

The key concept of object-oriented languages is a class. A class description contains fields, methods and constructors, to be explained below. Objects are instances of a class, having a state. Methods can change the state of an object. Often, the fields, methods and constructors of a class (together with their types, but without their bodies) are called the *interface* or signature of a class.

Fields, also known as instance variables, attributes or features, constitute the variable part of an object[1]. The values of the fields of an object at any point in time, completely characterise the state of an object. In JAVA fields are of a primitive type (*e.g.* integer or float) or they are references to objects. The objects that are referenced by a field are often called component objects. In some object-oriented languages, in particular SMALLTALK, everything is an object, and thus fields are always references to objects.

The methods of a class, also known as members or (functional) features, represent the computations that can be done on instances of that class. A method is like a procedure in a standard imperative language, with the scope limited to the object on which the method is called. This object is called the receiver object. The method body can refer directly to the fields and methods of the receiver object, but references to fields and methods in other objects are always made via a reference to their containing object. Such calls, denoted as *e.g.* `o.m()` are called qualified calls. In the case, the object `o` is the receiving object for the method `m()`.

The constructors of a class are used for creating new instances of a class. When a new instance is to be created, the constructor is called to perform the required initialisation action.

[1] In fact, the situation is more complicated, since static variables are shared by all instances of a class, but we ignore this, since it is not relevant for the ideas explained in this thesis.

Often, the constructor can be left implicit in the program code. In that case, a default constructor is called, which allocates memory cells for the new object and set all the fields in this new object to their default values. There are other object-oriented languages where the implicit constructor only allocates space for the new object.

Classes as they have been described so far, only seem to be an abstraction mechanism, grouping data and methods together, like in a module. But object-oriented languages also allow programmers to reuse existing classes when defining new ones. A new class `B` can be declared to extend an existing class `A`. This is also called: `B` inherits from `A`, `B` is a (direct) subclass of `A`, or `A` is a (direct) superclass of `B`. In this case, subclass `B` inherits all the fields and methods of superclass `A`. These fields and methods are immediately available in the subclass – no new implementation has to be given. This implies that all objects that are instances of class `B` can receive all the calls that objects in `A` can receive (but need not have the same behaviour). Therefore, everywhere an instance of superclass `A` is expected, an instance of subclass `B` can be used. This is often referred to as subtype polymorphism. If a variable is declared to refer to a class `A`, then at run-time it may contain references to instances of any subclass of `A` (including `A` itself). Therefore, a distinction has to be made between the static or declared type of a variable and its run-time type. In this thesis only single inheritance is considered, *i.e.* every subclass extends only one (direct) superclass.

In many object-oriented languages, including JAVA, it is the case that if no superclass is denoted explicitly (indicated by the keyword `extends`), a class implicitly inherits from the class `Object`. This class `Object` describes the basic functionality of every object (in the case of JAVA it implements for example an equality operation and a clone operation).

One of the crucial features of object-orientation is the possibility to override (or redefine) methods in subclasses: in a subclass, a new implementation of a method can be given. Suppose that class `A` has a method `m`, and class `B` inherits from `A`, but overrides `m`. Suppose that we have a variable `x` that is declared to belong to class `A`, and `x.m()` is called. Now, it depends on the run-time type of `x` which method implementation is actually executed. If `x` is an object in class `A`, then the old implementation of `m` is executed, but if `x` is in class `B`, then the new, redefining implementation is executed. This mechanism, where the actually executed method implementation depends on the run-time type of the receiving object is called late binding, also known as dynamic binding or dynamic method lookup.

Object-oriented languages differ in how they deal with redeclaring fields in subclasses. In some languages this is forbidden. In JAVA, it is allowed to redeclare a field in a subclass. The field in the superclass is then said to be hidden, because it cannot be accessed directly from the subclass anymore (except with an explicit call to the super class). If a field is used in a qualified call, *e.g.* `x.i` the decision which field is actually used is based on the declared (static) type of the object. Field lookup is thus independent of the run-time type of the object.

Within an object two special expressions can be used: `this` and `super`. The `this` expression always returns a reference to the current object. The `super` expression can be used to explicitly call an overridden method or hidden field of the superclass.

Many object-oriented languages allow a class to be not fully implemented, *i.e.* the implementation of some of the methods is still open. Nevertheless, these methods can be called in other methods. Such classes are usually called abstract classes. The methods without implementations are called abstract methods. Subclasses of the abstract class only have to give implementations of the abstract methods to make a concrete class (but of course they are allowed to override already implemented methods). Typically, the non-abstract methods in an

abstract class contain calls to the abstract methods. In a concrete class, extending such an abstract class, the appropriate implementations of these methods will be found via the late binding mechanism. A variant of abstract classes are so-called (class) interfaces. Interfaces only declare methods, they do not give any implementation. Thus, the method declarations in a class interface only describe what can be done, but not how it is done. Typically this is used to describe data structures, like sets; to lookup a value in a set of values, it is irrelevant to know how these values are actually stored. Implementations of the methods declared in a class interface are given in classes which `implement` the interface. For one class interface, several (different) implementations can be given. In this way, interfaces provide a means for abstraction in JAVA.

Chapter 2

A semantics for Java

This chapter presents a semantics for (sequential) JAVA. This presentation is divided into two parts: the first part (Sections 2.2 – 2.5) describes the basics of the semantics, *i.e.* the semantics for all forms of statement, which are the building blocks of the programming language. The semantics for these building blocks only has to be described once, and then can be reused over and over again in reasoning about arbitrary programs. This part contains the representation of statements and expressions, the representation of types, the memory model and all the other basic ingredients of the JAVA language. This collection of basic definitions is called the *semantic prelude*.

The second part (Section 2.6) describes the semantics of classes and objects. This semantics is captured in a translation from JAVA classes to our type theory, which generates for each class appropriate definitions, describing the meaning of that particular class. Coalgebras are used to represent classes. Appropriate manipulations of coalgebras allow us to model typical object-oriented behaviour, such as inheritance and overriding. Although the translation pattern is fixed, the outcome, *i.e.* the generated theories, are different for each JAVA class. The explanation is given in such a way that the translation pattern should become clear.

One class gives rise to a large collection of definitions and rewrite rules. Therefore, a compiler is developed which actually performs this translation (and generates logical theories in the input languages of the theorem provers PVS [ORR+96] and ISABELLE [Pau94]; see also Chapter 3). When reading about the translation from JAVA classes to type theory it is good to bear in mind that this translation is performed mechanically, a user gets all definitions by the compiler and only has to apply the reasoning. After the generation of the PVS or ISABELLE theories, loading the semantic prelude and the generated theories in the theorem prover allows reasoning about the JAVA classes, within the theorem prover.

One of the things that is typical for describing the semantics of a (real) programming language, is that many features of the language have to be made explicit. In the program code there are several things that are implicit. For example: if a class only has a default constructor, it does not need to be mentioned explicitly. However, when describing the meaning of this class formally, the constructor has to be mentioned explicitly. In this chapter we will encounter several examples of this process of making implicit language construct explicitly represented.

The semantics for JAVA presented in this chapter is formulated with the idea of program verification in mind. This means that many definitions are spelled out completely, because this improves the efficiency of the program verification process. If the JAVA semantics would be written down for different purposes, *e.g.* to prove meta properties about the language JAVA, such as type safety, different choices probably would have been made. In such verifications it

pays off to find common abstractions in different functions, because it makes the verification of the properties of the language easier. Examples of such abstraction in terms of a monad can be found in [JP00b].

The semantics below is described in a simple type theory and higher order logic, which can be seen as a common abstraction from the type theories and logics of both PVS and ISABELLE/HOL[1]. Using this general type theory and logic means that we can stay away from the peculiarities of PVS and ISABELLE and make this work more accessible to readers unfamiliar with these formalisms.

JAVA is a complete, complex programming language with many different features. Therefore, we concentrate on a part of the JAVA semantics and leave other topics as future work. Topics that are not discussed here, but are covered by the full semantics are:

- Recursive methods
- Exception handling
- Static fields and methods
- Access modifiers (usually handled statically by the compiler)

There are still other language features of which it is future work to describe their semantics.

- Inner classes are not handled by our semantics yet, but this should not be too hard; it only involves a lot of bureaucracy.
- For the time being we abstract away from precise number representation, for example we do not deal with integer bounds, and range and precision of floating point numbers. Incorporating this requires some care, to ensure that no problems occur in theorem proving.
- Incorporation of threads is still future work.

This chapter is organised as follows. Section 2.1 describes the simple type theory that we use to describe the JAVA semantics. Sections 2.2, 2.3, 2.4 and 2.5 describe the semantic prelude: the semantics of primitive types, references, statements, expressions and the underlying memory model. Section 2.6 describes the semantics of classes. The chapter concludes with conclusions and related work.

2.1 A simple type theory

This section introduces the type theory that we use to describe the JAVA semantics in the next sections. The terms in this type theory are used to form formulas in higher order logic. These higher order logic formulas are used later to denote (required) properties of JAVA programs.

Our type theory is a simple type theory with types built up from:

- type variables α, β, ...,

[1]Certain aspects of PVS and ISABELLE/HOL are incompatible, like the type parameters in PVS versus type polymorphism in ISABELLE/HOL, so that the type theory and logic that is used is not really in the intersection. But with some good will it should be clear how to translate the constructions that are presented into the particular languages of these proof tools. See Chapter 3 for a detailed explanation.

- type constants like nat, int, bool, string *etc.*,
- the recursive type constructor list,
- exponent types $\sigma \to \tau$,
- labeled product (or record) types $[\,\mathsf{lab}_1 : \sigma_1, \ldots, \mathsf{lab}_n : \sigma_n\,]$, and
- labeled coproduct (or variant) types $\{\,\mathsf{lab}_1 : \sigma_1 \mid \ldots \mid \mathsf{lab}_n : \sigma_n\,\}$,

for given types $\sigma, \tau, \sigma_1, \ldots, \sigma_n$, and with all lab_i distinct. Terms are the inhabitants of these types. For each type we present the relevant terms and operations.

For the type constructor list, the functions nil and cons are used as constructors and head and tail as destructors, such that

– TYPE THEORY

$$\forall l : \mathsf{list}[\alpha].\, l \neq \mathsf{nil} \supset \mathsf{cons}(\mathsf{head}\, l, \mathsf{tail}\, l) = l$$

There is an operator # on lists, returning the length of a list. Also, there exists a function every, which takes a predicate P and a list and returns true if all the elements in the list satisfy P.

For exponent types the standard notations for lambda abstraction $\lambda x : \sigma.\, M$ and application NL are used. In the sequel an *update* operation

– TYPE THEORY

$$f \text{ WITH } (i = N)$$

for exponent types is used, as abbreviation of the following function.

– TYPE THEORY

$$\lambda x : \sigma.\, \mathsf{IF}\ x = i\ \mathsf{THEN}\ N\ \mathsf{ELSE}\ f\, x$$

This operation satisfies the following, obvious, equations.

– TYPE THEORY

$$\begin{aligned} (f \text{ WITH } (i = N))\, i &= N \\ (f \text{ WITH } (i = N))\, j &= f\, j \vee i = j \end{aligned}$$

Given terms $M_i : \sigma_i$, the labeled tuple $(\,\mathsf{lab}_1 = M_1, \ldots, \mathsf{lab}_n = M_n\,)$ inhabits the labeled product type $[\,\mathsf{lab}_1 : \sigma_1, \ldots, \mathsf{lab}_n : \sigma_n\,]$. Given a term $N = (\,\mathsf{lab}_1 = M_1, \ldots, \mathsf{lab}_n = M_n\,)$ in this product, $N.\mathsf{lab}_i$ is written for the selection term returning M_i.

The Cartesian product type is a special instance of the labeled product type, with labels $\pi_1, \ldots, \pi_n$. We use the more usual notation $(M_1, \ldots, M_n) : \sigma_1 \times \ldots \times \sigma_n$ as an abbreviation for $(\,\pi_1 = M_1, \ldots, \pi_n = M_n\,) : [\pi_1 : \sigma_1, \ldots, \pi_n : \sigma_n]$.

Labeled products satisfy the following β- and η-conversions, precisely defining the behaviour of tupleing and selection.

– TYPE THEORY

$$(\,\mathsf{lab}_1 = M_1, \ldots, \mathsf{lab}_n = M_n\,).\mathsf{lab}_i \stackrel{\beta}{=} M_i$$
$$(\,\mathsf{lab}_1 = N.\mathsf{lab}_1, \ldots, \mathsf{lab}_n = N.\mathsf{lab}_n\,) \stackrel{\eta}{=} N$$

Also for labeled products an *update* operation is defined.

– TYPE THEORY

$$M \text{ WITH } (\mathsf{lab}_i = N)$$

which abbreviates the following labeled tuple.

– TYPE THEORY

$$\begin{array}{l}(\,\mathsf{lab}_1 = M.\mathsf{lab}_1,\\ \quad\vdots,\\ \quad \mathsf{lab}_{i-1} = M.\mathsf{lab}_{i-1},\\ \quad \mathsf{lab}_i = N,\\ \quad \mathsf{lab}_{i+1} = M.\mathsf{lab}_{i+1},\\ \quad\vdots,\\ \quad \mathsf{lab}_n = M.\mathsf{lab}_n\,)\end{array}$$

This update operation satisfies the following equations.

– TYPE THEORY

$$\begin{array}{lcl}(M \text{ WITH } (\mathsf{lab}_i = N)).\mathsf{lab}_i & = & N\\ (M \text{ WITH } (\mathsf{lab}_i = N)).\mathsf{lab}_j & = & M.\mathsf{lab}_j \;\vee\; i = j\end{array}$$

For a term $M : \sigma_i$ there is a tagged term $\mathsf{lab}_i\, M$, inhabiting a labeled coproduct type (or variant type) $\{\,\mathsf{lab}_1 : \sigma_1 \mid \ldots \mid \mathsf{lab}_n : \sigma_n\,\}$. Given a term N in this coproduct type, and given also n terms $L_i(x_i) : \tau$, each containing a free variable $x_i : \sigma_i$, there is a case term

– TYPE THEORY

$$\text{CASE } N \text{ OF } \{\,\mathsf{lab}_1\, x_1 \mapsto L_1(x_1) \mid \ldots \mid \mathsf{lab}_n\, x_n \mapsto L_n(x_n)\,\}$$

of type τ, satisfying the following (β)- and (η)-conversions (where $E[M/N]$ denotes E with all (free) occurrences of N substituted by M).

– TYPE THEORY

$$\begin{array}{lcl}\text{CASE } \mathsf{lab}_i\, M \text{ OF } \{ & & \\ \quad \mid \mathsf{lab}_1\, x_1 \mapsto L_1(x_1) & \stackrel{\beta}{=} & L_i[M/x_i]\\ \quad\vdots & & \\ \quad \mid \mathsf{lab}_n\, x_n \mapsto L_n(x_n)\,\} & & \\ \text{CASE } N \text{ OF } \{ & & \\ \quad \mid \mathsf{lab}_1\, x_1 \mapsto L[\mathsf{lab}_1 x_1/y] & \stackrel{\eta}{=} & L[N/y]\\ \quad\vdots & & \\ \quad \mid \mathsf{lab}_n\, x_n \mapsto L[\mathsf{lab}_n x_n/y]\,\} & & \end{array}$$

New types can be introduced via definitions, as in:

$$\mathsf{lift}[\alpha] \,:\, \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \{\, \mathsf{bot}\colon \mathsf{unit} \mid \mathsf{up}\colon \alpha \,\}$$

where unit is the empty product type [ ]. This lift type constructor adds a bottom element to an arbitrary type α. It is isomorphic with $1+\alpha$ where 1 is a one-element set and $+$ is disjoint union. It is frequently used in the sequel when modelling partial functions from α to β as functions of type $\alpha \to \mathsf{lift}[\beta]$.

Using the CASE construct, functions on lift can be defined, *e.g* the predicate defined? which is false for the bottom element.

– TYPE THEORY

$$\begin{array}{l} l\colon \mathsf{lift}[\alpha] \;\vdash \\ \quad \mathsf{defined?}(l) \;:\; \mathsf{bool} \stackrel{\mathrm{def}}{=} \\ \qquad \mathsf{CASE}\ l\ \mathsf{OF}\ \{ \\ \qquad\quad \mid \mathsf{bot} \mapsto \mathsf{false} \\ \qquad\quad \mid \mathsf{up}\, a \mapsto \mathsf{true}\ \} \end{array}$$

To denote properties about the (translated) JAVA programs, a higher order logic is introduced. Formulas in this higher order logic are terms of type bool. The connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), $\neg$ (negation, used with rules of classical logic) and constants true and false are used, together with the (typed) quantifiers $\forall x : \sigma.\, \varphi$ and $\exists x : \sigma.\, \varphi$, for a formula φ. There is a conditional term IF φ THEN M ELSE N, for terms M, N of the same type, satisfying the following equations.

– TYPE THEORY

$$\begin{array}{rcl} \mathsf{IF\ true\ THEN}\ M\ \mathsf{ELSE}\ N & \stackrel{\beta}{=} & M \\ \mathsf{IF\ false\ THEN}\ M \mathsf{ELSE}\ N & \stackrel{\beta}{=} & N \\ \mathsf{IF}\ \varphi\ \mathsf{THEN}\ L[\mathsf{true}/z] \mathsf{ELSE}\ L[\mathsf{false}/z] & \stackrel{\eta}{=} & L[\varphi/z] \end{array}$$

Notice that, instead of this conditional term, also a CASE distinction on the type bool can be used.

There is LET function, which can be used as abbreviation in definitions. It satisfies the following equations.

– TYPE THEORY

$$\mathsf{LET}\ x = E_1\ \mathsf{IN}\ E_2 \;\stackrel{\beta}{=}\; E_2[E_1/x]$$

Also a choice operator $\varepsilon x : \sigma.\, \varphi(x)$ exists, yielding a term of type σ, satisfying the following properties.

– TYPE THEORY

$$\begin{array}{l} \varphi(\varepsilon x : \sigma.\, \varphi\, x) \\ (\varepsilon x : \sigma.\, x = a) \stackrel{\eta}{=} a \end{array}$$

We shall use inductive definitions (over the types nat and list[α]), and also reason with the standard induction principle.

Sometimes we write comments in our type-theoretic definitions, to clarify a particular case. Comments are preceded by the symbol //.

All these language constructs are present in the specification languages of both PVS and ISABELLE/HOL. Thus, all type theoretic definitions that are given below, can be described in PVS and ISABELLE/HOL.

2.2 Java's primitive types and reference types

The primitive types in JAVA are:

```
byte, short, int, long, char, float, double, boolean
```

The first five of these are the so-called integral types. They have definite ranges in JAVA (*e.g.* `int` from -2147483648 to 2147483647). For all of these the existence of corresponding type constants byte, short, int, long, char, float, double and bool in our type theory is assumed[2].

Variables of reference type in JAVA refer to objects and arrays. The semantics of references is related to the memory model (Section 2.5). A reference may be `null`, indicating that it does not refer to anything. A non-null reference is a pointer to a memory location (in type MemLoc).

– TYPE THEORY

$$\mathsf{RefType} : \mathsf{TYPE} \stackrel{\text{def}}{=} \{\,\mathsf{null}: \mathsf{unit} \mid \mathsf{ref}: \mathsf{MemLoc}\}$$

The exact definition (and meaning) of the type MemLoc will be explained in Section 2.5. What is important here, is to notice that all references in JAVA (both to objects and to arrays) are translated in type theory to values of type RefType. Thus, given a reference `a` to an object in a class `A` and a reference `b` to an object in a subclass `B` of `A`, the assignment `a = b` is translated as a replacement of the reference to `a` by the reference to `b`. Since both are inhabitants of RefType, this is well-typed. If `b` has run-time type `B`, then so will `a` after the assignment.

2.3 Statements and expressions as state transformers

In classical program semantics the assumption is that statements will either terminate normally, resulting in a successor state, or will not terminate at all, see *e.g.* [Bak80, Chapter 3] or [Rey98, Section 2.2]. In the latter case one also says that the statement hangs, typically because of a non-terminating loop. Hence, statements may be understood as partial functions from states to states. First we shall use Self as a type variable representing the global state space. Later, in Section 2.5 a type OM is introduced, which describes a concrete state space. Then, the type

[2]One can take for example the type of integers $\ldots, -2, -1, 0, 1, 2, \ldots$ for the integral types, and the type of real numbers for the floating point types double and float, ignoring ranges and precision. As mentioned on page 10 it is still future work to include this.

variable Self will be instantiated with OM, but as long the details from OM are not needed, we prefer to use Self for abstraction. Statements can be seen as "state transformer" functions over Self

$$\mathsf{Self} \longrightarrow \mathsf{lift}[\mathsf{Self}] \quad (= 1 + \mathsf{Self})$$

This classical view of statements turns out to be inadequate for reasoning about JAVA programs. JAVA statements may hang, or terminate normally (like above), but they may additionally "terminate abruptly" (see *e.g.* [GJSB00, AG97]). Abrupt termination may be caused by an exception (typically a division by 0), a return, a break or a continue (inside a loop). Abrupt (or abnormal) termination is fundamentally different from non-termination: abnormalities affect the control flow of the program, but this effect can be temporary, because the abnormality may be caught at some later stage, whereas recovery from non-termination is impossible. Abnormalities can both be thrown and be caught, basically via re-arranging coproduct options. Constructs for both throwing and catching are described in type theory (see Section 2.4.2). Abrupt termination affects the flow of control: once it arises, all subsequent statements are ignored, until the abnormality is caught, see the definition of composition ";" in Section 2.4.1. From that moment on, the program executes normally again.

Abrupt termination requires a modification of the standard semantics of statements and expressions, resulting in a failure semantics, as for example in [Rey98, Section 5.1]. Therefore, in our approach, statements are modeled as more general state transformer functions

$$\mathsf{Self} \longrightarrow 1 + \mathsf{Self} + \mathsf{StatAbn}$$

where StatAbn (for Statement Abnormal, representing all the abnormalities that can be thrown by statements) forms a new alternative, which itself can be subdivided into four parts:

$$\mathsf{StatAbn} = \mathsf{Exception} + \mathsf{Return} + \mathsf{Break} + \mathsf{Continue}$$

These four constituents of StatAbn typically consist of a state in Self together with some extra information. An exception abnormality consists of a state together with a reference to an exception object. The reference is represented as an element of RefType, which is described above. A return abnormality only consists of a (tagged) state, and break and continue abnormalities consist of a state, possibly with a label. This structure of the codomain of our JAVA state transformer function is captured formally in a variant type StatResult (see Figure 2.1).

In classical semantics, expressions are viewed as functions

$$\mathsf{Self} \longrightarrow \mathsf{Out}$$

where Out is the type of the result of the expression. This view is not quite adequate for our purposes, because it does not involve non-termination, abrupt termination or side-effects: an expression in JAVA may hang, terminate normally or terminate abruptly. If it terminates normally, it produces an output result (of the type of the expression) together with a state (since it may have a side-effect). If it terminates abruptly, this can only be because of an exception (and not because of a break, continue, or return, see [GJSB00, §15.5]). Hence a JAVA expression of type Out is (in our view) a function of the form:

$$\mathsf{Self} \longrightarrow 1 + (\mathsf{Self} \times \mathsf{Out}) + \mathsf{ExprAbn}$$

– TYPE THEORY

```
StatResult[Self] : TYPE =def
    { hang   : unit
    | norm   : Self
    | abnorm : StatAbn[Self] }

ExprResult[Self, Out] : TYPE =def
    { hang   : unit
    | norm   : [ ns: Self, res: Out ]
    | abnorm : ExprAbn[Self] }

StatAbn[Self] : TYPE =def
    { excp   : [ es: Self, ex: RefType ]
    | rtrn   : Self
    | break  : [ bs: Self, blab: lift[string] ]
    | cont   : [ cs: Self, clab: lift[string] ] }

ExprAbn[Self] : TYPE =def
    [ es: Self, ex: RefType ]
```

Figure 2.1: The types StatResult and ExprResult

The first alternative (1) captures the situation where an expression hangs. The second alternative (Self $\times$ Out) occurs when an expression terminates normally, resulting in a successor state together with an output result. The final alternative (ExprAbn) describes abrupt termination – because of an exception – for expressions. Again, this is captured by a suitable variant type ExprResult in Figure 2.1.

To summarise, in our semantics, statements are modeled as functions from Self to StatResult[Self], and expressions as functions from Self to ExprResult[Self, Out], for the appropriate result type Out.

This abstract representation of statements and expressions as "one entry/multi-exit" functions (terminology of [Chr84]) forms the basis for the work presented here. It is used to give a (denotational) meaning to basic programming constructs like composition, if-then-else, and while.

To conclude, there is one technicality that deserves attention. Sometimes an expression has to be transformed into a statement, which is only a matter of forgetting the result of the expression. However, in our semantics this transformation has to be done explicitly, using a function E2S.

– TYPE THEORY

```
e : Self → ExprResult[Self, Out] ⊢

    E2S(e) : Self → StatResult[Self] =def
        λx : Self. CASE e x OF {
                     | hang ↦ hang
                     | norm y ↦ norm(y.ns)
                     | abnorm a ↦ abnorm(excp(es = a.es, ex = a.ex)) }
```

In the last line an expression abnormality (an exception) is transformed into a statement abnormality.

2.4 Java statements and expressions

Based on the types representing statements and expressions, the semantics for various program constructs can be described, closely following the JAVA language specification [GJSB00]. The notation $[\![S]\!]$ is used to denote the interpretation (translation) of the JAVA statement or expression S in type theory.

This section first discusses the semantics of several "standard" non-looping JAVA statements. (`skip`, statement composition and `if`). It is shown how their semantics relates to the JAVA language specification [GJSB00]. Subsequently, the translation of abruptly terminating statements (like `return` and `break`) into type theory is explained, followed by a discussion of the semantics of the loop statements (as `while` and `for`). Finally, the semantics of JAVA expressions is discussed.

2.4.1 Basic, non-looping statements

Skip

The most basic statement is the empty statement `skip`, which always terminates normally, returning its initial state. It is translated as follows:

$$[\![\texttt{skip}]\!] = \mathsf{skip}$$

where skip is defined in type theory as:

– TYPE THEORY

$$\begin{array}{l} \mathsf{skip} \ : \ \mathsf{Self} \to \mathsf{StatResult[Self]} \stackrel{\text{def}}{=} \\ \quad \lambda x : \mathsf{Self}.\, \mathsf{norm}\, x \end{array}$$

Statement composition

The sequential statement composition operator `;` is translated by the type-theoretic function ";" as follows.

$$[\![\texttt{s;t}]\!] = [\![\texttt{s}]\!] \,;\, [\![\texttt{t}]\!]$$

The function ";" has the following definition in type theory.

– TYPE THEORY

$$\begin{array}{l} s, t : \mathsf{Self} \to \mathsf{StatResult[Self]} \ \vdash \\ \quad (s\,;t) \ : \ \mathsf{Self} \to \mathsf{StatResult[Self]} \stackrel{\text{def}}{=} \\ \quad\quad \lambda x : \mathsf{Self}.\ \ \mathsf{CASE}\ s\,x\ \mathsf{OF}\ \{ \\ \quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\ \quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto t\, y \\ \quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a\ \} \end{array}$$

Thus if statement s terminates normally in state x, resulting in a next state y, then $(s\,;t)\,x$ is $t\,y$. And if s hangs or terminates abruptly in state x, then $(s\,;t)\,x$ is $s\,x$ and t is not executed. This binary operation ";" forms a monoid with the skip statement defined above.

– TYPE THEORY

$$\text{Assuming } s, t, u : \mathsf{Self} \to \mathsf{StatResult[Self]}$$

$$\begin{aligned} \mathsf{skip}\,;s &= s \\ s\,;\mathsf{skip} &= s \\ (s\,;t)\,;u &= s\,;(t\,;u) \end{aligned}$$

If-then-else

As mentioned above, all JAVA language constructs are formalised in a similar way, following closely the JAVA language specification [GJSB00]. As an example, the translation of the `if ... else` statement is considered in more detail. This statement is translated as follows.

$$[\![\texttt{if (cond) S else T}]\!] \stackrel{\text{def}}{=} \mathsf{IF\text{-}THEN\text{-}ELSE}([\![\texttt{cond}]\!])([\![\texttt{S}]\!])([\![\texttt{T}]\!])$$

To define the type-theoretic function IF-THEN-ELSE the description of the `if ... else` statements in [GJSB00, §14.8] is considered.

> ## 14.8 The `if` statement
>
> The `if` statement allows conditional execution of a statement or a conditional choice of two statements, executing one or the other but not both.
>
> ...
>
> *IfThenElseStatement*:
>
> `if` (*Expression*) *Statement* `else` *Statement*
>
> ...
>
> ### 14.8.2 The `if-then-else` Statement
>
> An `if-then-else` statement is executed by first evaluating the *Expression*. If evaluation of the *Expression* completes abruptly for some reason, then the `if-then-else` statement completes abruptly for the same reason. Otherwise, execution continues by making a choice based on the result value:
>
> - If the value is `true`, then the first contained *Statement* (the one before the `else` keyword) is executed; the `if-then-else` statement completes normally only if execution of that statement completes normally.
> - If the value is `false`, then the second contained *Statement* (the one after the `else` keyword) is executed; the `if-then-else` statement completes normally only if execution of that statement completes normally.

Following closely this description, we get the next definition of IF-THEN-ELSE in type theory.

– TYPE THEORY

$$
\begin{array}{l}
c : \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{bool}],\ s, t : \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}] \ \vdash \\
\quad \mathsf{IF\text{-}THEN\text{-}ELSE}(c)(s)(t) \ : \ \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}] \stackrel{\text{def}}{=} \\
\quad\quad \lambda x : \mathsf{Self}.\ \ \mathsf{CASE}\ c\,x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\quad\quad\quad\quad \mid \mathsf{norm}\,y \mapsto \ \ \mathsf{CASE}\ y.\mathsf{res}\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{true} \mapsto \ \ \mathsf{CASE}\ s\ (y.\mathsf{ns})\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\,z \mapsto \mathsf{norm}\,z \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\,b \mapsto \mathsf{abnorm}\,b\ \} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{false} \mapsto \ \ \mathsf{CASE}\ t\ (y.\mathsf{ns})\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\,z \mapsto \mathsf{norm}\,z \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\,b \mapsto \mathsf{abnorm}\,b\ \} \\
\quad\quad\quad\quad \mid \mathsf{abnorm}\,a \mapsto \mathsf{abnorm}(\mathsf{excp}(\mathsf{es} = a.\mathsf{es}, \mathsf{ex} = a.\mathsf{ex}))\ \}
\end{array}
$$

Using η-conversion on the CASE construct, this simplifies to the following definition.

– TYPE THEORY

$$
\begin{array}{l}
c : \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{bool}],\ s, t : \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}] \ \vdash \\
\quad \mathsf{IF\text{-}THEN\text{-}ELSE}(c)(s)(t) \ : \ \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}] \stackrel{\text{def}}{=} \\
\quad\quad \lambda x : \mathsf{Self}.\ \ \mathsf{CASE}\ c\,x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\quad\quad\quad\quad \mid \mathsf{norm}\,y \mapsto \ \ \mathsf{IF}\ y.\mathsf{res} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{THEN}\ s\ (y.\mathsf{ns}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{ELSE}\ t\ (y.\mathsf{ns}) \\
\quad\quad\quad\quad \mid \mathsf{abnorm}\,a \mapsto \mathsf{abnorm}(\mathsf{excp}(\mathsf{es} = a.\mathsf{es}, \mathsf{ex} = a.\mathsf{ex}))\ \}
\end{array}
$$

Notice that all our translations incorporate the argument-first, left-to-right evaluation strategy of JAVA, see [GJSB00, §§15.6].

2.4.2 Abruptly terminating statements

This section discusses the semantics of abruptly terminating statements. This differs from the semantics of the "normal" statements in the previous section, since it does not only involve the formalisation of *throwing* abnormalities, *e.g.* formalising `return`, but also the formalisation of *catching* abnormalities. In a JAVA program, this is done implicitly, but in our semantics, this becomes explicit. Thus, appropriate functions have to be defined and explicitly inserted in the type-theoretic description of a JAVA program. Here we consider abnormalities because of `return`'s, `break`'s and `continue`'s. Throwing and catching exceptions uses the same mechanism, but it also involves the semantics of object creation (see Section 2.6.11) and the `instanceof` operation (see Section 2.6.10). It is not discussed in this thesis, for more information see [Jac00].

Return

When a `return` statement is executed, the program immediately exits from the current method. A `return` statement in a non-void method has an expression argument; this expression is evaluated and returned as the result of the method. The translation of the JAVA `return` statement (without argument) is,

$$[\![\texttt{return}]\!] = \mathsf{RETURN}$$

where RETURN is defined in type theory as:

– TYPE THEORY

$$\mathsf{RETURN} : \mathsf{Self} \to \mathsf{StatResult[Self]} \stackrel{\text{def}}{=} \lambda x : \mathsf{Self}.\, \mathsf{abnorm}(\mathsf{rtrn}\, x)$$

This statement produces an abnormal state, which can be caught at the end of a method body. The translation of a `return` statement with argument is similar, but more subtle. First the value of the expression is stored in a special local variable, and then the state becomes abnormal, via the above RETURN.

$$[\![\texttt{return expr}]\!] \stackrel{\text{def}}{=} [\![\texttt{ret\_var = expr}]\!]\,;\mathsf{RETURN}$$

To recover from a return abnormality, we use functions CATCH-STAT-RETURN and CATCH-EXPR-RETURN, respectively. In our translation of JAVA programs, a function CATCH-STAT-RETURN is wrapped around every method body that returns `void`. First the method body is executed. This may result in an abnormal state, because of a return. In that case the function CATCH-STAT-RETURN turns the state back to normal again. Otherwise, it leaves the state unchanged.

– TYPE THEORY

$$\begin{array}{l} s : \mathsf{Self} \to \mathsf{StatResult[Self]} \vdash \\ \quad \mathsf{CATCH\text{-}STAT\text{-}RETURN}(s) : \mathsf{Self} \to \mathsf{StatResult[Self]} \stackrel{\text{def}}{=} \\ \quad\quad \lambda x : \mathsf{Self}.\ \mathsf{CASE}\ s\, x\ \mathsf{OF}\ \{ \\ \quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\ \quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto \mathsf{norm}\, y \\ \quad\quad\quad\quad \mid \mathsf{abnorm}\, a \mapsto\ \mathsf{CASE}\ a\ \mathsf{OF}\ \{ \\ \quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{excp}\, e \mapsto \mathsf{abnorm}(\mathsf{excp}\, e) \\ \quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{rtrn}\, z \mapsto \mathsf{norm}\, z \\ \quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{break}\, b \mapsto \mathsf{abnorm}(\mathsf{break}\, b) \\ \quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{cont}\, c \mapsto \mathsf{abnorm}(\mathsf{cont}\, c)\ \}\ \} \end{array}$$

RETURN and CATCH-STAT-RETURN satisfy the following equations.

– TYPE THEORY

$$\begin{array}{l} \text{Assuming } s : \mathsf{Self} \to \mathsf{StatResult[Self]} \\ \qquad\qquad \mathsf{RETURN}\,;s = \mathsf{RETURN} \\ \qquad \mathsf{CATCH\text{-}STAT\text{-}RETURN(RETURN)} = \mathsf{skip} \end{array}$$

If a method returns a value, a function CATCH-EXPR-RETURN is used, instead of CATCH-STAT-RETURN. Recall that the result value of a method is stored in a special variable. The function CATCH-EXPR-RETURN possibly turns the state back to normal and, in that case, returns the output held by this special variable.

– TYPE THEORY

$$
\begin{array}{l}
s : \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}],\ v : \mathsf{Out} \vdash \\
\quad \mathsf{CATCH\text{-}EXPR\text{-}RETURN}(s)(v) \ :\ \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{Out}] \stackrel{\text{def}}{=} \\
\quad\quad \lambda x : \mathsf{Self}.\ \ \mathsf{CASE}\ s\,x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\quad\quad\quad\quad \mid \mathsf{norm}\,y \mapsto \mathsf{hang}/\!/\ \text{should not happen} \\
\quad\quad\quad\quad \mid \mathsf{abnorm}\,a \mapsto\ \ \mathsf{CASE}\ a\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{excp}\,e \mapsto \mathsf{abnorm}(\mathsf{excp}\,e) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{rtrn}\,z \mapsto \mathsf{norm}\,(\mathsf{ns} = z, \mathsf{res} = v) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{break}\,b \mapsto \mathsf{hang} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{cont}\,c \mapsto \mathsf{hang}\ \}\ \}
\end{array}
$$

Notice that for a correct JAVA program it is required that a method body that returns a value, always throws a return abnormality (unless an exception occurred). Thus, in contrast to CATCH-STAT-RETURN, the function CATCH-EXPR-RETURN returns hang if s is normal or abnormal because of a break or continue.

Break

A `break` statement can be used to exit from any block. If a `break` statement is labeled, it exits the block with that label. A `break` statement with label `lab` must occur inside a (nested) block with label `lab`, so that it cannot be used as an arbitrary goto. Unlabeled `break` statements exit the innermost `switch`, `for`, `while` or `do` statement. The JAVA language requires that there is always a point where the break abnormality is caught.

A JAVA `break` statement is translated as

$$
\begin{array}{rcl}
[\![\texttt{break}]\!] & \stackrel{\text{def}}{=} & \mathsf{BREAK} \\
[\![\texttt{break label}]\!] & \stackrel{\text{def}}{=} & \mathsf{BREAK\text{-}LABEL}(\text{“}\texttt{label}\text{”})
\end{array}
$$

where BREAK and BREAK-LABEL(l), for l : string, are defined as functions with type Self → StatResult[Self]:

– TYPE THEORY

$$
\begin{array}{rcl}
\mathsf{BREAK} & \stackrel{\text{def}}{=} & \lambda x : \mathsf{Self}.\ \mathsf{abnorm}(\mathsf{break}(\mathsf{bs} = x, \mathsf{blab} = \mathsf{bot})) \\
\mathsf{BREAK\text{-}LABEL}(l) & \stackrel{\text{def}}{=} & \lambda x : \mathsf{Self}.\ \mathsf{abnorm}(\mathsf{break}(\mathsf{bs} = x, \mathsf{blab} = \mathsf{up}\,(l)))
\end{array}
$$

Figure 2.2 shows an associated function CATCH-BREAK which turns abnormal states, because of breaks with the appropriate label, back into normal states.

In the JAVA translation [JBH+98] every labeled block is enclosed with CATCH-BREAK applied to the appropriate label:

$$
[\![\texttt{label:body}]\!] \stackrel{\text{def}}{=} \mathsf{CATCH\text{-}BREAK}(\mathsf{up}(\text{“}\texttt{label}\text{”}))([\![\texttt{body}]\!])
$$

– TYPE THEORY

ll : lift[string], s : Self $\to$ StatResult[Self] $\vdash$

CATCH-BREAK(ll)(s) : Self $\to$ StatResult[Self] $\stackrel{\text{def}}{=}$

λx : Self. CASE $s\,x$ OF {
| hang $\mapsto$ hang
| norm $y \mapsto$ norm y
| abnorm $a \mapsto$ CASE a OF {
| excp $e \mapsto$ abnorm(excp e)
| rtrn $z \mapsto$ abnorm(rtrn z)
| break $b \mapsto$ IF b.blab $= ll$
THEN norm(b.bs)
ELSE abnorm(break b)
| cont $c \mapsto$ abnorm(cont c) } }

Figure 2.2: Definition of CATCH-BREAK

As unlabeled breaks exit the innermost `switch`, `while`, `for` and `do` statement, all these statements are enclosed with CATCH-BREAK applied to bot. It is not possible to catch labeled and unlabeled breaks within the same CATCH-BREAK. As an example, consider the following (silly) fragment of JAVA code.

– JAVA

```
while (true) {
  lab : { x = y;
          if (c) {break};
          x = 4;
        };
        y = 3;
}
```

Notice that, because the `break` is unlabeled, the `while` statement is exited, if the `break` is executed. If the break would be labeled with label `lab`, only the statement `x = 4` would have been skipped and normal execution would have resumed at the statement `y = 3`.

Translating this into type theory, gives the following expression (using WHILE as the type-theoretic description for the `while` statements, as defined in Section 2.4.3).

– TYPE THEORY

```
CATCH-BREAK(bot)(
  WHILE(bot)([[true]])(
    CATCH-BREAK(up(lab))(
      [[x = y]];
      IF-THEN([[c]])(BREAK);
      [[x = 4]]);
    [[y = 3]]))
```

If CATCH-BREAK(up(`lab`)) would also catch unlabeled breaks, this fragment would have a different behaviour than the corresponding JAVA fragment.

Similar properties as for the `return` statement hold for the functions BREAK, BREAK-LABEL and CATCH-BREAK.

– TYPE THEORY

Assuming s : Self $\to$ StatResult[Self]
l, m : string

$$\begin{aligned}
\text{BREAK} ; s &= \text{BREAK} \\
\text{BREAK-LABEL}(l) ; s &= \text{BREAK-LABEL}(l) \\
\text{CATCH-BREAK(bot)(BREAK)} &= \text{skip} \\
\text{CATCH-BREAK(up}(l)\text{)(BREAK)} &= \text{BREAK} \\
\text{CATCH-BREAK(bot)(BREAK-LABEL}(l)\text{)} &= \text{BREAK-LABEL}(l) \\
\text{CATCH-BREAK(up}(l)\text{)(BREAK-LABEL}(l)\text{)} &= \text{skip} \\
\text{CATCH-BREAK(up}(m)\text{)(BREAK-LABEL}(l)\text{)} &= \text{BREAK-LABEL}(l) \ \vee \ l = m
\end{aligned}$$

Continue

Within loop statements (`while`, `do` and `for`) a `continue` statement can occur. The effect is that control skips the rest of the loop's body and starts re-evaluating (the update statement, in a `for` loop, and) the Boolean expression which controls the loop. A `continue` statement can be labeled, so that the `continue` applies to the correspondingly labeled loop, and not to the innermost one.

A JAVA `continue` statement is translated as

$$\begin{aligned}
[\![\texttt{continue}]\!] &\stackrel{\text{def}}{=} \text{CONTINUE} \\
[\![\texttt{continue label}]\!] &\stackrel{\text{def}}{=} \text{CONTINUE-LABEL}(\text{“}\texttt{label}\text{”})
\end{aligned}$$

where CONTINUE and CONTINUE-LABEL(l), for l : string, are defined as functions Self $\to$ StatResult[Self]:

– TYPE THEORY

$$\begin{aligned}
\text{CONTINUE} &\stackrel{\text{def}}{=} \lambda x : \text{Self. abnorm(cont(cs} = x, \text{clab} = \text{bot))} \\
\text{CONTINUE-LABEL}(l) &\stackrel{\text{def}}{=} \lambda x : \text{Self. abnorm(cont(cs} = x, \text{clab} = \text{up}\ (l)))
\end{aligned}$$

A function CATCH-CONTINUE is defined, which turns abnormal states that are caused by a `continue` statement, back into normal states. This function is used to describe the semantics of looping statements; after every iteration of the loop body, possible `continue`'s are caught, after which normal execution resumes, see Section 2.4.3.

Unlabeled `continue`'s always should be caught immediately, by the innermost enclosing loop, while a labeled `continue` is caught by the appropriately labeled loop. In contrast to CATCH-BREAK, the function CATCH-CONTINUE will catch both labeled and unlabeled `continue` abnormalities.

– TYPE THEORY

ll : lift[string], s : Self → StatResult[Self] ⊢

CATCH-CONTINUE(ll)(s) : Self → StatResult[Self] $\stackrel{\text{def}}{=}$

λx : Self. CASE $s\,x$ OF {
| hang ↦ hang
| norm y ↦ norm y
| abnorm a ↦ CASE a OF {
| excp e ↦ abnorm(excp e)
| rtrn z ↦ abnorm(rtrn z)
| break b ↦ abnorm(break b)
| cont c ↦ IF c.clab = ll ∨ c.clab = bot
THEN norm(c.cs)
ELSE abnorm(cont c) } }

The functions CONTINUE, CONTINUE-LABEL and CATCH-CONTINUE satisfy similar properties as BREAK, BREAK-LABEL and CATCH-BREAK. Notice that for expressions e : Self → ExprResult[Self, Out] also the following holds.

– TYPE THEORY

Assuming s : Self → StatResult[Self]
e : Self → ExprResult[Self, bool]
ll : lift[string]

E2S(e) ; CATCH-CONTINUE(ll)(s) = CATCH-CONTINUE(ll)(E2S(e) ; s)

A similar property holds for CATCH-STAT-RETURN, CATCH-EXPR-RETURN, and CATCH-BREAK, but we explicitly state it here, since in this form it is relevant to the semantic description of looping statements below.

2.4.3 Looping statements

JAVA has three different loop statements: `while`, `do` and `for`. This section describes in detail the semantics of the `while` statement. Given this, the translation of the other looping statements is straightforward.

To describe the semantics of the looping statements, special care is needed, because in type theory, all functions have to be total, while in JAVA looping statements might not terminate. In that case, evaluation of the statement in type theory should result in hang. Therefore, it first is decided whether the loop terminates (either normally or abruptly), and then an appropriate result is returned[3].

Iteration

The core of the semantics of the looping statements is the function iterate, which iterates a statement. Its definition is based on the semantics for skip and statement composition.

[3]The function that checks whether the loop terminates does not have an executable definition, thus we did not solve the halting problem.

– TYPE THEORY

$$s : \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}], n : \mathsf{nat} \vdash$$

$$\mathsf{iterate}(s, n) \ : \ \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}] \stackrel{\text{def}}{=}$$

$$\lambda x : \mathsf{Self}.\ \ \mathsf{IF}\ n = 0$$
$$\mathsf{THEN\ skip}$$
$$\mathsf{ELSE\ iterate}(s, n-1)\ ;\ s$$

This function satisfies the following properties.

– TYPE THEORY

$$\text{Assuming } s : \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}], n, m : \mathsf{nat}$$

$$\begin{aligned}
\mathsf{iterate}(s, 0) &= \mathsf{skip} \\
\mathsf{iterate}(s, 1) &= s \\
s\ ;\ \mathsf{iterate}(s, n) &= \mathsf{iterate}(s, n+1) = \mathsf{iterate}(s, n)\ ;\ s \\
\mathsf{iterate}(s, m+n) &= \mathsf{iterate}(s, m)\ ;\ \mathsf{iterate}(s, n) \\
\mathsf{iterate}(s, m*n) &= \mathsf{iterate}(\mathsf{iterate}(s, m), n)
\end{aligned}$$

While

The JAVA `while` statement is translated as follows.

$$\begin{aligned}
[\![\texttt{while (cond) \{body\}}]\!] &\stackrel{\text{def}}{=} \mathsf{CATCH\text{-}BREAK(bot)} \\
&\qquad (\mathsf{WHILE(bot)}([\![\texttt{cond}]\!])([\![\texttt{body}]\!])) \\
[\![\texttt{lab:while (cond) \{body\}}]\!] &\stackrel{\text{def}}{=} \mathsf{CATCH\text{-}BREAK(up}(\text{“}\texttt{lab}\text{”}\mathsf{))} \\
&\qquad (\mathsf{CATCH\text{-}BREAK(bot)} \\
&\qquad\quad (\mathsf{WHILE(up}(\text{“}\texttt{lab}\text{”}\mathsf{))}([\![\texttt{cond}]\!])([\![\texttt{body}]\!])))
\end{aligned}$$

The surrounding CATCH-BREAK(bot) makes sure that the while loop terminates normally if an unlabeled break occurs in its body. If a labeled break occurs in the loop, there must be a correspondingly labeled (block) statement surrounding this break statement. This ensures that the labeled break is caught.

Figure 2.6 shows the definition of WHILE in type theory, making use of auxiliary predicates NoStops, NormalStopNumber? and AbnormalStopNumber? from Figures 2.3, 2.4 and 2.5. The function iterate described above, is applied to the composite statement

$$\mathsf{E2S}(\mathit{cond})\ ;\ \mathsf{CATCH\text{-}CONTINUE}(\mathit{lift\_label})(\mathit{body})$$

where lift_label is either bot or up(“`lab`”). Below, this statement will be referred to as the iteration body. It first evaluates the condition (for its side-effect, discarding its result), and then evaluates the statement, making sure that occurrences of a continue (with appropriate label) in this statement are caught. The function NoStops tells for every number n whether the iteration body will be executed at least n times (which means that the condition is true after m iterations, for $m < n$, and iterating the iteration body n times terminates normally).

– TYPE THEORY

c: Self $\to$ ExprResult[Self, bool], s: Self $\to$ StatResult[Self], x: Self $\vdash$

NoStops(c, s, x) : nat $\to$ [result: bool, state: Self] $\stackrel{\text{def}}{=}$

λn: nat. IF $\forall m$: nat. $m < n \supset$
CASE iterate(E2S(c) ; $s, m)$ x OF {
| hang $\mapsto$ false
| norm $y \mapsto$ CASE c y OF {
| hang $\mapsto$ false
| norm $z \mapsto$ z.res
| abnorm $b \mapsto$ false }
| abnorm $a \mapsto$ false }
THEN CASE iterate(E2S(c) ; $s, n)$ x OF {
| hang $\mapsto$ (result $=$ false, state $=$ x)
| norm $y \mapsto$ (result $=$ true, state $=$ y)
| abnorm $a \mapsto$ (result $=$ false, state $=$ x) }
ELSE (result $=$ false, state $=$ x)

Figure 2.3: Auxiliary function NoStops

– TYPE THEORY

c: Self $\to$ ExprResult[Self, bool], s: Self $\to$ StatResult[Self], x: Self $\vdash$

NormalStopNumber?(c, s, x) : nat $\to$ bool $\stackrel{\text{def}}{=}$

λn: nat. (NoStops(c, s, x) n).result $\wedge$
CASE c ((NoStops(c, s, x) n).state) OF {
| hang $\mapsto$ false
| norm $y \mapsto \neg(y$.res)
| abnorm $a \mapsto$ false}

Figure 2.4: Auxiliary function NormalStopNumber?

– TYPE THEORY

c: Self $\to$ ExprResult[Self, bool], s: Self $\to$ StatResult[Self], x: Self $\vdash$

AbnormalStopNumber?(c, s, x) : nat $\to$ bool $\stackrel{\text{def}}{=}$

λn: nat. (NoStops(c, s, x) n).result $\wedge$
CASE (E2S(c) ; $s)$ ((NoStops(c, s, x) n).state) OF {
| hang $\mapsto$ false
| norm $y \mapsto$ false
| abnorm $a \mapsto$ true}

Figure 2.5: Auxiliary function AbnormalStopNumber?

$$
\begin{array}{l}
ll : \mathsf{lift[string]}, c : \mathsf{Self} \to \mathsf{ExprResult[Self, bool]}, s : \mathsf{Self} \to \mathsf{StatResult[Self]} \vdash \\
\quad \mathsf{WHILE}(ll)(c)(s) \ : \ \mathsf{Self} \to \mathsf{StatResult[Self]} \stackrel{\text{def}}{=} \\
\qquad \lambda x : \mathsf{Self}.\ \ \mathsf{LET}\ \mathit{iter\_body} = \mathsf{E2S}(c)\,;\mathsf{CATCH\text{-}CONTINUE}(ll)(s), \\
\qquad\qquad\qquad \mathit{NormalStopSet} = \\
\qquad\qquad\qquad\quad \mathsf{NormalStopNumber?}(c, \mathsf{CATCH\text{-}CONTINUE}(ll)(s), x), \\
\qquad\qquad\qquad \mathit{AbnormalStopSet} = \\
\qquad\qquad\qquad\quad \mathsf{AbnormalStopNumber?}(c, \mathsf{CATCH\text{-}CONTINUE}(ll)(s), x) \\
\qquad\qquad \mathsf{IN}\ \mathsf{IF}\ \exists n : \mathsf{nat}.\, \mathit{NormalStopSet}\ n \\
\qquad\qquad\quad \mathsf{THEN}\ \big(\mathsf{iterate}(\mathit{iter\_body}, \varepsilon n : \mathsf{nat}.\, \mathit{NormalStopSet}\ n)\,;\,\mathsf{E2S}(c)\big)\ x \\
\qquad\qquad\quad \mathsf{ELSIF}\ \exists n : \mathsf{nat}.\, \mathit{AbnormalStopSet}\ n \\
\qquad\qquad\quad \mathsf{THEN}\ \big(\mathsf{iterate}(\mathit{iter\_body}, \varepsilon n : \mathsf{nat}.\, \mathit{AbnormalStopSet}\ n)\,;\,\mathit{iter\_body}\big)\ x \\
\qquad\qquad\quad \mathsf{ELSE}\ \mathsf{hang}
\end{array}
$$

Figure 2.6: WHILE in type theory, using definitions from Figures 2.3, 2.4 and 2.5

The sets NormalStopNumber? and AbnormalStopNumber?(Figures 2.4 and 2.5) characterise the point where the loop will terminate in the next iteration, either because the condition becomes false, resulting in normal termination of the loop, or because an abnormality occurs in the iteration body, resulting in abnormal termination of the loop. From the definitions it follows that if NormalStopNumber? or AbnormalStopNumber? is non-empty, then it is a singleton. And if both are non-empty, then the number in NormalStopNumber? is at most the number in AbnormalStopNumber?. Therefore, the WHILE function first checks if NormalStopNumber? is non-empty, and subsequently if AbnormalStopNumber? is non-empty. In both cases, the iteration body is executed the appropriate number of times, so that the loop will terminate in the next iteration. In the case of normal termination this is followed by an additional execution of the condition (for its side-effect), and in the case of abnormal termination this is followed by an execution of the iteration body, resulting in abrupt termination. If both sets NormalStopNumber? and AbnormalStopNumber? are empty, the loop will never terminate (normally or abruptly), thus hang is returned. Basically, this definition makes WHILE a least fixed point, see [JP00b] for details. As the definition of WHILE is not executable, we can not prove properties about it using automatic rewriting. In order to be enable reasoning about looping statements in a convenient way, Chapter 5 presents a Hoare logic tailored to JAVA.

This definition satisfies the following equation (where IF-THEN is defined similar to IF-THEN-ELSE on page 19).

$$
\begin{array}{l}
\text{Assuming } s : \mathsf{Self} \to \mathsf{StatResult[Self]} \\
\qquad\qquad e : \mathsf{Self} \to \mathsf{ExprResult[Self, bool]} \\
\qquad\qquad ll : \mathsf{lift[string]} \\
\qquad \mathsf{WHILE}(ll)(c)(s) \ = \mathsf{IF\text{-}THEN}(c)(\mathsf{CATCH\text{-}CONTINUE}(ll)(s)\,;\,\mathsf{WHILE}(ll)(c)(s))
\end{array}
$$

Do

The `do` statement in JAVA always executes its body at least once. It is interpreted via a function DO.

$$\begin{aligned}
[\![\texttt{do s while (c)}]\!] &\stackrel{\text{def}}{=} \text{CATCH-BREAK(bot)} \\
&\quad (\text{DO(bot)}([\![\texttt{c}]\!])([\![\texttt{s}]\!])) \\
[\![\texttt{lab:do s while (c)}]\!] &\stackrel{\text{def}}{=} \text{CATCH-BREAK(up(“}\texttt{lab}\text{”))} \\
&\quad (\text{CATCH-BREAK(bot)} \\
&\quad\quad (\text{DO(up(“}\texttt{lab}\text{”))}([\![\texttt{c}]\!])([\![\texttt{s}]\!])))
\end{aligned}$$

This function DO is defined in terms of the WHILE statement in type theory:

– TYPE THEORY ———————————————————————————

$$\begin{aligned}
&ll : \text{lift[string]}, c : \text{Self} \to \text{ExprResult[Self, bool]}, s : \text{Self} \to \text{StatResult[Self]} \vdash \\
&\quad \text{DO}(ll)(c)(s) \; : \; \text{Self} \to \text{StatResult[Self]} \stackrel{\text{def}}{=} \\
&\quad\quad \text{CATCH-CONTINUE}(ll)(s) \; ; \; \text{WHILE}(ll)(c)(s)
\end{aligned}$$

For

The semantics of the `for` statement is similar to that of the `while` statement. It is translated into type theory as follows.

$$\begin{aligned}
&[\![\texttt{for(init; cond; update)\{body\}}]\!] \stackrel{\text{def}}{=} \\
&\quad [\![\texttt{init}]\!]; \\
&\quad \text{CATCH-BREAK(bot)} \\
&\quad\quad (\text{FOR(bot)}([\![\texttt{cond}]\!])([\![\texttt{update}]\!])([\![\texttt{body}]\!])) \\
&\\
&[\![\texttt{lab:for(init; cond; update)\{body\}}]\!] \stackrel{\text{def}}{=} \\
&\quad [\![\texttt{init}]\!]; \\
&\quad \text{CATCH-BREAK(up(“}\texttt{lab}\text{”))} \\
&\quad\quad (\text{CATCH-BREAK(bot)} \\
&\quad\quad\quad (\text{FOR(up(“}\texttt{lab}\text{”))}([\![\texttt{cond}]\!])([\![\texttt{update}]\!])([\![\texttt{body}]\!]))
\end{aligned}$$

A `for` statement has four (possibly empty) components: (1) an initialisation statement `init`, (2) a condition `cond`, (3) an update statement `update`, consisting of so-called *expression statements* only, *i.e.* expressions which are executed for their side-effects, discarding their results, (4) a body `body`. The initialisation statement is executed exactly once. As long as the condition holds, the body is executed, followed by the update statement. Even if a continue (with appropriate label) occurred in the body, the update statement will still be executed at the end of the iteration. Notice that, since the update statement consists of expressions only, this

– TYPE THEORY

ll : lift[string],
c : Self $\to$ ExprResult[Self, bool],
u : Self $\to$ StatResult[Self],
s : Self $\to$ StatResult[Self] $\vdash$

FOR(ll)(c)(u)(s) : Self $\to$ StatResult[Self] $\stackrel{\text{def}}{=}$
λx : Self. LET *iter_body* = E2S(c) ; CATCH-CONTINUE(ll)(s) ; u,
NormalStopSet =
NormalStopNumber?(c, CATCH-CONTINUE(ll)(s) ; u, x)
AbnormalStopSet =
AbnormalStopNumber?(c, CATCH-CONTINUE(ll)(s) ; u, x)
IN
IF $\exists n$: nat. *NormalStopSet* n
THEN (iterate(*iter_body*, εn : nat. *NormalStopSet* n) ;
E2S(c)) x
ELSIF $\exists n$: nat. *AbnormalStopSet* n
THEN (iterate(*iter_body*, εn : nat. *AbnormalStopSet* n)
iter_body) x
ELSE hang

Figure 2.7: Definition of FOR

will never terminate abruptly because of a `continue` (or a `break` or `return`). Compared to the `while` statement, a `for` statement has a slightly different iteration body, namely:

E2S(*cond*) ; CATCH-CONTINUE(*lift_label*)(*body*) ; *update*

where *lift_label* is either bot or up("`lab`").

The type-theoretic definition of FOR in Figure 2.7 incorporates these differences.

Notice that WHILE and FOR can be expressed in each other[4] as follows.

– TYPE THEORY

Assuming s : Self $\to$ StatResult[Self]
c : Self $\to$ ExprResult[Self, Out]
ll : lift[string]

WHILE(ll)(c)(s) = FOR(ll)(c)(skip)(s)
FOR(ll)(c)(u)(s) = WHILE(ll)(c)(s ; u)

[4]Assuming that if u : Self$\to$ StatResult[Self] terminates abruptly, this is because of an exception. This is a reasonable assumption, because the update statement actually consists of *ExpressionStatements* only (see [GJSB00, §§14.12]), thus it will only terminate abruptly because of an exception.

2.4.4 Expressions

The semantics of expressions is described similar to the semantics of statements, following closely Chapter 15 of [GJSB00]. Some examples are given, to show the basic ideas.

Constant expressions

The most basic expression is the constant expression. For each type Out with inhabitant a : Out a constant expression const(a) is defined as:

– TYPE THEORY

$$a : \mathsf{Out} \vdash$$
$$\mathsf{const}(a) \ : \ \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{Out}] \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{Self}.\, \mathsf{norm}(\mathsf{ns} = x, \mathsf{res} = a)$$

Clearly, constant expressions have no side-effects.

This constant expression is used to translate JAVA literals, like `0`, `1.0`, `1.36d`, `true` *etc.*, as:

$$\begin{aligned}
[\![\texttt{0}]\!] &\stackrel{\text{def}}{=} \mathsf{const}(0) \ : \ \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{int}] \\
[\![\texttt{1.0}]\!] &\stackrel{\text{def}}{=} \mathsf{const}(10 * 10^{-1}) \ : \ \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{double}] \\
[\![\texttt{1.36d}]\!] &\stackrel{\text{def}}{=} \mathsf{const}(136 * 10^{-2}) \ : \ \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{double}] \\
[\![\texttt{true}]\!] &\stackrel{\text{def}}{=} \mathsf{const}(\mathsf{true}) \ : \ \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{bool}]
\end{aligned}$$

Notice that the following equation holds for const.

– TYPE THEORY

$$\text{Assuming } a : \mathsf{Out}$$
$$\mathsf{E2S}(\mathsf{const}(a)) \ = \ \mathsf{skip}$$

Expression composition

In JAVA, a programmer can use postfix operators for incrementing and decrementing, *e.g.* `i++` (see [GJSB00, §§15.13]). First the value of the variable is evaluated, then the value 1 is added to the value of the variable and the sum is stored back into the variable. The whole expression returns the value of the variable *before* addition. To translate this into type theory, a special expression composition ;; is needed which composes two expressions (namely the variable lookup and the assignment) and returns the result of the first expression[5].

Thus, *e.g* the JAVA postfix increment operator is translated as follows:

$$[\![\texttt{i++}]\!] \ = \ [\![\texttt{i}]\!] \,;;\, [\![\texttt{i = i + 1}]\!]$$

[5]Notice that prefix in- and decrement operators and assignment operations like `+=` all can be translated as simple assignments. *E.g.* `++i` and `i+=1` both are equal to `i=i+1`.

where the expression composition operation " ;; " is defined as follows in type theory.

– TYPE THEORY

e_1, e_2 : Self $\to$ ExprResult[Self, Out] $\vdash$

$(e_1 \;;;\; e_2)$: Self $\to$ ExprResult[Self, Out] $\stackrel{\text{def}}{=}$

λx : Self. CASE $e_1\, x$ OF {
| hang $\mapsto$ hang
| norm $y \mapsto$ CASE $e_2\, y$ OF {
| hang $\mapsto$ hang
| norm $z \mapsto$ norm(ns $= z$.ns, res $= y$.res)
| abnorm $b \mapsto$ abnorm b }
| abnorm $a \mapsto$ abnorm a }

Thus, first expression e_1 is evaluated. If this terminates normally, e_2 is evaluated in the result state produced by e_1. If this also terminates normally, the result value of expression e_1 is returned, together with the state produced by e_2.

This operation satisfies the following equations.

– TYPE THEORY

Assuming e_1, e_2, e_3 : Self $\to$ ExprResult[Self, Out], a : Out

$$\begin{array}{rcl} \text{E2S}(e_1 \;;;\; e_2) & = & \text{E2S}(e_1)\,;\,\text{E2S}(e_2) \\ e_1 \;;;\; (e_2 \;;;\; e_3) & = & (e_1 \;;;\; e_2) \;;;\; e_3 \\ e_1 \;;;\; \text{const}(a) & = & e_1 \end{array}$$

Binary operators

As a last example, the type-theoretic definition for addition is given. This definition is typical for the semantics of binary operators. Notice the left-to-right evaluation order and the incorporation of side-effects. First e_1 is evaluated. If this terminates normally, e_2 is evaluated in the result state produced by e_1. If this also terminates normally, the value of the addition is returned, together with the state produced by e_2.

– TYPE THEORY

e_1, e_2 : Self $\to$ ExprResult[Self, int] $\vdash$

$e_1 + e_2$: Self $\to$ ExprResult[Self, int] $\stackrel{\text{def}}{=}$

λx : Self. CASE $e_1\, x$ OF {
| hang $\mapsto$ hang
| norm $y \mapsto$
CASE e_2 (y.ns) OF {
| hang $\mapsto$ hang
| norm $z \mapsto$ norm(ns $= z$.ns, res $= y$.res $+ z$.res)
| abnorm $b \mapsto$ abnorm b }
| abnorm $a \mapsto$ abnorm a }

– TYPE THEORY

$$
\begin{array}{l}
\mathsf{ObjectCell} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\
\quad \big[\, \mathsf{bytes} : \mathsf{CellLoc} \to \mathsf{byte}, \\
\quad\ \ \mathsf{shorts} : \mathsf{CellLoc} \to \mathsf{short}, \\
\quad\ \ \mathsf{ints} : \mathsf{CellLoc} \to \mathsf{int}, \\
\quad\ \ \mathsf{longs} : \mathsf{CellLoc} \to \mathsf{long}, \\
\quad\ \ \mathsf{chars} : \mathsf{CellLoc} \to \mathsf{char}, \\
\quad\ \ \mathsf{floats} : \mathsf{CellLoc} \to \mathsf{float}, \\
\quad\ \ \mathsf{doubles} : \mathsf{CellLoc} \to \mathsf{double}, \\
\quad\ \ \mathsf{booleans} : \mathsf{CellLoc} \to \mathsf{bool}, \\
\quad\ \ \mathsf{refs} : \mathsf{CellLoc} \to \mathsf{RefType}, \\
\quad\ \ \mathsf{type} : \mathsf{string}, \\
\quad\ \ \mathsf{dimlen} : [\, \mathsf{dim} : \mathsf{nat}, \mathsf{len} : \mathsf{nat} \,] \,\big]
\end{array}
$$

Figure 2.8: The type ObjectCell, representing single memory cells

Notice that for binary operators on numbers, also a more abstract definition could be given, which is parametrised with $op : \mathsf{int} \times \mathsf{int} \to \mathsf{int}$. Addition would then be defined as this abstract function, instantiated with the $+$ operation. It can easily be shown that this addition operation satisfies its usual properties, *e.g.* addition is commutative and associative and has 0 as its identity element.

– TYPE THEORY

$$
\begin{array}{l}
\text{Assuming } e, e_1, e_2, e_3 : \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{int}] \\
\qquad\qquad
\begin{array}{rcl}
e_1 + e_2 & = & e_2 + e_1 \\
e_1 + (e_2 + e_3) & = & (e_1 + e_2) + e_3 \\
e + \mathsf{const}(0) & = & e \\
\mathsf{const}(0) + e & = & e
\end{array}
\end{array}
$$

For unary operators, similar definitions are given, which first evaluate the argument and in the case of normal termination apply the operation.

2.5 The memory model

After discussing the semantics of JAVA statements and expressions, we now focus on a more low-level aspect of the formalisation, namely the underlying memory model that is used.

This section starts by defining memory cells for storing JAVA objects and arrays. They are used to build up the main memory for storing arbitrarily many of such items. This object memory OM comes with various operations for reading and writing. More information on the memory model is given in [BHJP00]. From now on, statements are understood as partial functions from OM to OM, thus the type variable Self is instantiated with OM.

2.5.1 Memory cells

A single memory cell can store the contents of all the fields from a single object of an arbitrary class. The (translated) types that the fields of objects can have are limited to byte, short, int, long, char, float, double, bool and RefType (as defined in Section 2.2). Therefore a cell should be able to store elements of all these types. The number of fields for a particular type is not bounded, so infinitely many are incorporated in a memory cell. Additionally, a cell has an entry type of type string and an entry dimlen, which is a pair of natural numbers. If the cell contents represent an object the type entry indicates its run-time type, and if it represents an array, type indicates its elementtype. In the latter case, the dimlen entry denotes the length and dimension of the array. For ordinary objects, the length and dimension are set to 0, thus denoting that the cell does not denote an array. The type of memory cell is depicted in Figure 2.8. The type CellLoc that is used, is defined as follows.

– TYPE THEORY

$$\mathsf{CellLoc}\colon \mathsf{Type} \stackrel{\text{def}}{=} \mathsf{nat}$$

Our memory is organised in such a way that each memory location points to a memory cell, and each cell location to a position inside the cell.

Storing an object from a class with, for instance, two integer fields and one Boolean field in a memory cell is done by (only) using the first two values (at 0 and at 1) of the function ints: CellLoc $\to$ int and (only) the first value (at 0) of the function booleans: CellLoc $\to$ bool. Other values of these and other functions in the object cell are irrelevant. The LOOP compiler attributes these cell locations to (static) fields of a class, local variables and parameters. The actual cell locations are hidden away from the user. More information on the link between fields and cell locations is given in [BHJP00].

An empty memory cell is defined with Java's default values (see [GJSB00, §§ 4.5.4]) for primitive types and reference types. The type entry is set to the empty string, and the dimension and length are set to 0.

– TYPE THEORY

$$\begin{aligned}\mathsf{EmptyObjectCell}\colon \mathsf{ObjectCell} \stackrel{\text{def}}{=} \big(&\mathsf{bytes} = \lambda n\colon \mathsf{CellLoc}.\,0,\\ &\mathsf{shorts} = \lambda n\colon \mathsf{CellLoc}.\,0,\\ &\mathsf{ints} = \lambda n\colon \mathsf{CellLoc}.\,0,\\ &\mathsf{longs} = \lambda n\colon \mathsf{CellLoc}.\,0,\\ &\mathsf{chars} = \lambda n\colon \mathsf{CellLoc}.\,0,\\ &\mathsf{floats} = \lambda n\colon \mathsf{CellLoc}.\,0,\\ &\mathsf{doubles} = \lambda n\colon \mathsf{CellLoc}.\,0,\\ &\mathsf{booleans} = \lambda n\colon \mathsf{CellLoc}.\,\mathsf{false},\\ &\mathsf{refs} = \lambda n\colon \mathsf{CellLoc}.\,\mathsf{null},\\ &\mathsf{type} = \text{""},\\ &\mathsf{dimlen} = (\,\mathsf{dim} = 0, \mathsf{len} = 0\,)\big)\end{aligned}$$

Storing an empty object cell at a particular memory location guarantees that all field values stored there get default values.

2.5.2 Object memory

Object cells form the main ingredient of the new type OM representing all memory. It has a heap, stack and static part, for storing the contents of respectively instance variables, local variables and parameters of method invocations, and static (also called class) variables:

– TYPE THEORY

$$\begin{array}{l} \mathsf{OM} : \mathsf{TYPE} \stackrel{\text{def}}{=} \\ \quad [\, \mathsf{heapmem}: \mathsf{MemLoc} \to \mathsf{ObjectCell}, \\ \quad\ \mathsf{heaptop}: \mathsf{MemLoc}, \\ \quad\ \mathsf{stackmem}: \mathsf{MemLoc} \to \mathsf{ObjectCell}, \\ \quad\ \mathsf{stacktop}: \mathsf{MemLoc}, \\ \quad\ \mathsf{staticmem}: \mathsf{MemLoc} \to [\, \mathsf{initialised}: \mathsf{bool}, \mathsf{staticcell}: \mathsf{ObjectCell} \,] \,] \end{array}$$

The type MemLoc is defined as follows.

– TYPE THEORY

$$\mathsf{MemLoc}: \mathsf{Type} \stackrel{\text{def}}{=} \mathsf{nat}$$

The entry heaptop (resp. stacktop) indicates the next free (unused) memory location on the heap (resp. stack). The LOOP compiler assigns locations (in the static memory) to classes with static fields. At such locations a Boolean initialised tells whether static initialisation has taken place for this class. One must keep track of this because static initialisation should be performed at most once. The JAVA virtual machine performs initialisation at compile-time (or load-time). However, in our semantics, static initialisation is performed when the class is used for the first time. Abstracting away from memory limitations, this does not affect the observable behaviour of the system.

Reading and writing in the object memory

Accessing a specific value in an object memory x : OM, either for reading or for writing, involves the following ingredients:
– an indication of which part of memory (heap, stack, static),
– a memory location (in MemLoc),
– the type of the value and
– a cell location (in CellLoc) giving the offset in the cell.
These ingredients are combined in the following variant type for memory addressing.

– TYPE THEORY

$$\begin{array}{l} \mathsf{MemAdr} : \mathsf{TYPE} \stackrel{\text{def}}{=} \\ \quad \{\, \mathsf{heap}: [\, \mathsf{ml}: \mathsf{MemLoc}, \mathsf{cl}: \mathsf{CellLoc} \,] \\ \quad\ \mid \mathsf{stack}: [\, \mathsf{ml}: \mathsf{MemLoc}, \mathsf{cl}: \mathsf{CellLoc} \,] \\ \quad\ \mid \mathsf{static}: [\, \mathsf{ml}: \mathsf{MemLoc}, \mathsf{cl}: \mathsf{CellLoc} \,] \,\} \end{array}$$

For each type typ from the collection of types byte, short, int, long, char, float, double, bool and RefType occurring in object cells (see the definition of ObjectCell), there are two operations:

$$\begin{array}{c} \mathsf{get\_typ}: \mathsf{MemAdr} \to \mathsf{OM} \to \mathsf{typ} \\ \mathsf{put\_typ}: \mathsf{MemAdr} \to \mathsf{OM} \to \mathsf{typ} \to \mathsf{OM} \end{array}$$

These functions are described in detail only for typ = byte; the other cases are similar. Reading from the memory is easy, as described in function get_byte.

– TYPE THEORY

$$\vdash \text{get\_byte} : \text{MemAdr} \to \text{OM} \to \text{byte} \stackrel{\text{def}}{=}$$
$$\lambda m : \text{MemAdr}.\ \lambda x : \text{OM}.$$
$$\text{CASE } m \text{ OF } \{$$
$$\quad | \text{ heap } \ell \mapsto ((x.\text{heapmem}(\ell.\text{ml})).\text{bytes})(\ell.\text{cl})$$
$$\quad | \text{ stack } \ell \mapsto ((x.\text{stackmem}(\ell.\text{ml})).\text{bytes})(\ell.\text{cl})$$
$$\quad | \text{ static } \ell \mapsto ((x.\text{staticmem}(\ell.\text{ml})).\text{staticcell.bytes})(\ell.\text{cl}) \}$$

The corresponding write-operation uses updates of records and also updates of functions; we combine this into one single 'WITH' operation.

– TYPE THEORY

$$\vdash \text{put\_byte} : \text{MemAdr} \to \text{OM} \to \text{byte} \to \text{OM} \stackrel{\text{def}}{=}$$
$$\lambda m : \text{MemAdr}.\ \lambda x : \text{OM}.\ \lambda u : \text{typ}.$$
$$\text{CASE } m \text{ OF } \{$$
$$\quad | \text{ heap } \ell \mapsto x \text{ WITH } [\ ((x.\text{heapmem}(\ell.\text{ml})).\text{bytes})(\ell.\text{cl}) = u\ ]$$
$$\quad | \text{ stack } \ell \mapsto x \text{ WITH } [\ ((x.\text{stackmem}(\ell.\text{ml})).\text{bytes})(\ell.\text{cl}) = u\ ]$$
$$\quad | \text{ static } \ell \mapsto x \text{ WITH}$$
$$\qquad [\ ((x.\text{staticmem}(\ell.\text{ml})).\text{staticcell.bytes})(\ell.\text{cl}) = u\ ] \}$$

Similar definitions get_type, get_dimlen, put_type and put_dimlen exist. The various get- and put-functions are related as follows.

– TYPE THEORY

$$\text{Assuming } m, n : \text{MemAdr}, x : \text{OM}, u : \text{byte}, v : \text{short}$$
$$\text{get\_byte } n\ (\text{put\_byte } m\ x\ u) = \text{IF } m = n \text{ THEN } u \text{ ELSE get\_byte } n\ x$$
$$\text{get\_byte } n\ (\text{put\_short } m\ x\ v) = \text{get\_byte } n\ x$$

Such equations are used for auto-rewriting: whenever these equations can be applied, the back-end proof-tool simplifies goals automatically.

2.5.3 Operations on references

Section 2.2 explained how reference types are formalised. Notice that in our formalisation, just as in JAVA, a reference points to some memory location in memory. Thus, this allows us to reason about aliasing. If two references are pointing to the same object, then changes to this object via one reference, are also visible via the other reference. As an example, consider the following JAVA classes.

```
class TheObject {

  int i;

}

class Aliasing {

  TheObject a;
  TheObject b;

  void m() {
    a = new TheObject();
    b = a;
    a.i = 3;
  }
}
```

After the method `m` is executed, `a` and `b` refer to the same object. The field `i` in this object is changed, via the reference `a`. Since `a` and `b` are aliases (because of the assignment `b = a`), this means that `b.i` also equals 3. This behaviour is captured by our formalisation. In the translation from JAVA classes to type-theoretic definitions (as discussed in Section 2.6) the LOOP compiler assigns memory locations to the fields of the translated objects. Suppose that an instance of `Aliasing` is stored at memory location p, with its fields linked to memory locations as follows.

a	heap(ml= p, cl= 0)
b	heap(ml= p, cl= 1)

The translated assignment `a = new TheObject();` first allocates and initialises memory on the heap for the new object, say at heaptop x, resulting in a new state y. Then it assigns a reference to this new object to `a`, by

$$\mathsf{put\_ref}(\mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 0))\ y\ (\mathsf{ref}\ (\mathsf{heaptop}\ x))$$

Say that this returns a state z. The next assignment `b = a` is then translated into the following operations on the memory.

$$\begin{array}{l} \mathsf{put\_ref}\ (\mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 1)) \\ \qquad z \\ \qquad (\mathsf{get\_ref}(\mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 0))\ z) \end{array}$$

Thus, the values on memory locations $\mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 0)$ and $\mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 1)$ are the same after this assignment. The last assignment `a.i = 3` changes the `i` field of the new object, following the reference to the object on $\mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 0)$. If subsequently `b.i` is accessed, the reference at $\mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 1)$ is followed, leading to the same object with field `i` equalling 3. That `b.i` equals 3 after execution of `m` can be proven (automatically) in our formalisation. Thus, the references `a` and `b` are aliases and changes via one reference are also visible via the other one.

Reference comparison

Based on the type RefType, operations on references can be formalised in type theory, *e.g.* testing for reference equality is translated as

$$[\![\texttt{r1 == r2}]\!] \stackrel{\text{def}}{=} [\![\texttt{r1}]\!] == [\![\texttt{r2}]\!]$$

where $==$ is defined in type theory, following [GJSB00, §§ 15.20.3] as follows.

– TYPE THEORY

$$\begin{array}{l} r_1, r_2 : \mathsf{OM} \to \mathsf{ExprResult[OM, RefType]} \vdash \\ \quad r_1 == r_2 \ : \ \mathsf{OM} \to \mathsf{ExprResult[OM, bool]} \stackrel{\text{def}}{=} \\ \quad\quad \lambda x : \mathsf{OM}. \\ \quad\quad\quad \mathsf{CASE}\, r_1\, x\, \mathsf{OF}\, \{ \\ \quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\ \quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto \\ \quad\quad\quad\quad\quad \mathsf{CASE}\, r_2\, (y.\mathsf{ns})\, \mathsf{OF}\, \{ \\ \quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\ \quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\, z \mapsto \\ \quad\quad\quad\quad\quad\quad\quad \mathsf{norm}\ \ (\mathsf{ns} = z.\mathsf{ns}, \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{res} = (y.\mathsf{res} = z.\mathsf{res}) \\ \quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\, b \mapsto \mathsf{abnorm}\, b \,\} \\ \quad\quad\quad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a \,\} \end{array}$$

The this expression

Given the memory location p of an object, a reference to this object can be created. This is used to formalise JAVA's `this` expression. A function this is defined, returning a reference to the object in which the expression is evaluated (see [§§15.7.2][GJSB00]).

The this function takes as argument the memory location at which the object is stored.

– TYPE THEORY

$$\begin{array}{l} p : \mathsf{MemLoc} \vdash \\ \quad \mathsf{this}(p) \ : \ \mathsf{OM} \to \mathsf{ExprResult[OM, RefType]} \stackrel{\text{def}}{=} \\ \quad\quad \lambda x : \mathsf{OM}.\, \mathsf{norm}\, (\, \mathsf{ns} = x, \mathsf{res} = \mathsf{ref}\, p\,) \end{array}$$

The `this` function can only be called from within a method or constructor body, and since these bodies are always parametrised with their memory location in memory (see Section 2.6.8) the necessary information is always available.

2.5.4 Operations on arrays

The modelling of arrays in our semantics is a typical example of how the object memory is used. Arrays are stored as references, pointing to a cell where the actual data is stored. In this cell the

entry type denotes the elementtype of the array (either a primitive type, *e.g.* `int` or `float`, or a reference type) and the entry dimlen denotes the length and dimension of the array.

For arrays of arrays, the dimension of the array is set to 2, and this generalises to n-dimensional arrays. The dimension information of arrays is used for type information, *e.g.* to check casts. Typical operations on arrays are array creation, lookup and assignment. The semantics of these operations is discussed below. It shows in more detail how the memory model is used. In a similar way, other operations, such as `array.length` are defined.

Array initialisation

Array creation expressions in JAVA are translated into a function new_array in type theory. In general, array initialisation is translated as follows:

$$[\![\texttt{new ClassName [expr1]...[exprn][] ... []}]\!] \stackrel{\text{def}}{=}$$
$$\mathsf{new\_array}(\texttt{"ClassName"})([\,[\![\texttt{expr1}]\!], \ldots, [\![\texttt{exprn}]\!], \mathsf{const}(0), \ldots, \mathsf{const}(0)\,])$$

where the number of const(0) expressions equals the number of unspecified dimensions in the array creation expression.

The type-theoretic function new_array is defined in Figure 2.9, using the auxiliary functions evaluate_expr_list and put_array_refs defined below. The function new_array first evaluates the index expressions, by using the function evaluate_expr_list. The list of index expressions cannot be empty, thus in our type-theoretic definition (which has to be total) we return something arbitrary in this case. For non-empty lists it is checked whether all index expressions are positive, and otherwise an exception is thrown. If all index expressions are positive, the array structure is set up by calling the function put_array_refs. This structure starts on the old heaptop (heaptop $(y.\mathsf{ns})$). After setting up the structure, the new heaptop is set past this structure, by using the function heaptop_inc. The type and dimlen entries of the memory cell at the old heaptop are set appropriately, and the state that is produced in this way is returned, together with a reference to the old heaptop, *i.e.* a reference to the newly created array.

The first auxiliary function that is used in the definition of new_array is evaluate_expr_list, defined in Figure 2.10, which takes a list of expressions and a state, and evaluates all these expressions. If the evaluation of all expressions terminates normally, a list with the results is returned. The expressions are evaluated from left to right, passing on the state to incorporate possible side-effects. This function is used to evaluate the expressions denoting the size of the array.

Notice that the result is only added to the list of results when the tail of the list has been evaluated. This ensures that the order of the results is the same as the order of the expressions in the arguments *exprs*.

The other auxiliary function is put_array_refs (Figure 2.11), which assigns correct values to the references, thus creating the structure for the array on the heap.

To understand this function we first look at an example. Suppose we call put_array_refs with $[2, 3, 4]$ as *bounds*, heaptop x for *cur_pos* and heaptop $x + 1$ for *next_free_pos* and the string `"int"` for *str*. The result of this call, creating the structure for a 3-dimensional array, is visualised (and simplified) in Figure 2.12.

The first column represents the refs entry of the object cell at heaptop x. Notice that the type and dimlen entry of this memory cell are not set by put_array_refs, but by the function

```
str : string, index_exprs : list[OM → ExprResult[OM, RefType]] ⊢
    new_array(str)(index_exprs) : OM → ExprResult[OM, RefType] ≝
        λx : OM.
            CASE evaluate_expr_list(nil, index_exprs) x OF {
              | hang ↦ hang
              | norm y ↦
              CASE y.res OF {
                | nil ↦ hang // should not happen
                | cons c ↦
                  IF every(λi : int. i ≥ 0)(y.res)
                  THEN ⟦new NegativeArraySizeException()⟧
                  ELSE  LET put_references =
                                put_array_refs (y.res)
                                               (y.ns,
                                                heaptop(y.ns),
                                                heaptop(y.ns) + 1,
                                                str)
                        IN
                        norm (ns = put_type
                                     (heaptop(y.ns))
                                     (put_dimlen
                                       (heaptop(y.ns))
                                       (heaptop_inc
                                         (put_references.state)
                                         (put_references.nfp−
                                           heaptop(y.ns)))
                                       (dim = #(index_exprs),
                                        len = c.head))
                                     str,
                              res = ref (heaptop(y.ns))
              | abnorm a ↦ abnorm(excp(es = a.es, ex = a.ex))
```

Figure 2.9: Function new_array

```
results: list[Out], exprs: list[Self → ExprResult[Self, Out]] ⊢
    evaluate_expr_list(results, exprs) : Self → ExprResult[Self, list[Out]] =def
        λx : OM.  CASE exprs OF {
                    | nil ↦ norm(ns = x, res = results)
                    | cons c ↦
                      CASE c.head x OF {
                          | hang ↦ hang
                          | norm y ↦
                            CASE evaluate_expr_list(results, c.tail)(y.ns) OF {
                                | hang ↦ hang
                                | norm z ↦ norm ( ns = z.ns,
                                                  res = cons ( head = y.res,
                                                               tail = z.res ) )
                                | abnorm b ↦ abnorm b }
                          | abnorm a ↦ abnorm a } }
```

Figure 2.10: Definition of evaluate_expr_list

new_array, after the whole structure has been created. The first two cells are occupied, containing references to heaptop $x + 1$ (the next free memory location) and heaptop $x + 5$. If the array that we are constructing is called a, then these references represent `a[0]` and `a[1]`. Later in the function new_array, after the call to put_array_refs, the type of this cell will be set to int, and the dimlen entry will be set to (dim $= 3$, len $= 2$).

The cells at heaptop $x + 1$ and heaptop $x + 5$ both have a type int, a length 3 and a dimension 2, since they are both representing 2-dimensional arrays of integers with size 3 by 4. The refs entry of the object cell at heaptop $x + 1$ contains references to the memory cells at heaptop $x + 2$, heaptop $x + 3$ and heaptop $x + 4$, with type $=$ int and dimlen $=$ (dim $= 1$, len $= 4$). Similarly for the refs entry at heaptop $x + 5$.

The recursive call of put_array_refs with *next_free_pos* is *e.g.* heaptop x + 2 will have *bounds* equal to [4] as argument, thus the tail of bounds is nil. The only effect of this recursive call is that an empty ObjectCell is put on the heap at this memory location. In this "clean" cell, the elements of the array can be stored at the appropriate places. In this case, where *str* = `"int"`, the elements will be stored in the ints entry of these object cells.

For example, the value of `a[0][1][2]` will be stored in the cell location ints(2) of the object cell at heaptop $x + 3$.

In general, the function put_array_refs is defined as follows. It takes a list of index values, which are all greater or equal than 0. If the list is empty we are done (actually this is never the case, since the function is always called on an non-empty list, see the definition of new_array above). If the list is a singleton list, this means that we are creating a one-dimensional array, thus no structure has to be build.

If the list is longer, say $[b_1, b_2, \ldots, b_n]$ with $n > 1$, the following happens. A function *put_array_refs_rec* is iterated b_1 times. In the first iteration this function puts a reference in heap(ml $=$ *cur_pos*, cl $= 0$) to a new cell, and recursively calls put_array_refs on this

```
bounds: list[nat], x : OM, cur_pos: MemLoc, next_free_pos: MemLoc, str: string ⊢
  put_array_refs(bounds)(x, cur_pos, next_free_pos, str) :
    [state: OM, nfp: MemLoc] =def
      CASE bounds OF {
        | nil ↦ (state = x, nfp = next_free_pos)
        | cons c ↦
          CASE c.tail OF {
            | nil ↦  (state = put_empty_heap x cur_pos,
                      nfp = next_free_pos)
            | cons d ↦
              LET put_references =
                (iterate
                  (λr : (state: OM, nfp: MemLoc, cellpos: CellLoc).
                    LET put_array_refs_rec =
                      put_array_refs (c.tail)
                                     (put_ref
                                       (heap(ml = cur_pos,
                                             cl = r.cellpos))
                                       (put_type (r.nfp)
                                                 (put_dimlen
                                                   (r.nfp)
                                                   (r.state)
                                                   (dim = #(c.tail),
                                                    len = d.head))
                                                 str)
                                       (ref (r.nfp)))
                                     (r.nfp) (r.nfp + 1) str
                    IN (state = put_array_refs_rec.state,
                        nfp = put_array_refs_rec.nfp,
                        cellpos = r.cellpos + 1))
                  (c.head)
                  (state = x, nfp = next_free_pos, cellpos = 0))
              IN (state = put_references.state,
                  nfp = put_references.nfp) } }
```

Figure 2.11: Function put_array_refs

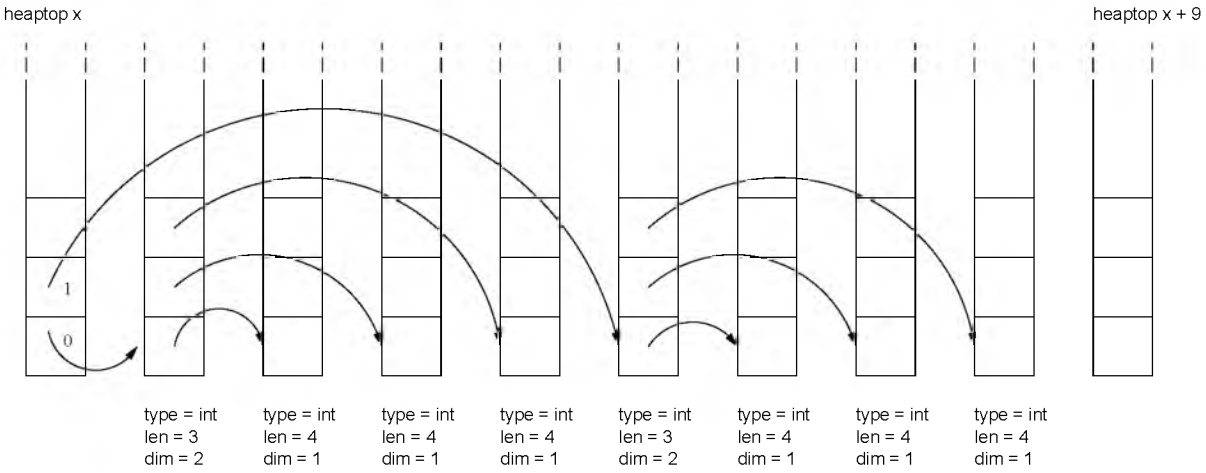

Figure 2.12: put_array_refs[2, 3, 4](heaptop x)(heaptop $x + 1$)(str)

new cell, with the list $[b_2, \ldots, b_n]$, and the memory location of this new cell as the current memory location argument. This recursive call creates the structure for the array with dimensions $[b_2, \ldots, b_n]$ and it returns the new state space and the next free memory location in memory.

Subsequently, the next iteration puts a reference at heap(ml $=$ *cur_pos*, cl $=$ 1) to a new cell at the heap at the next free memory location, thus past the structure that has been built in the first recursive call. Again, a recursive call to put_array_refs is made, and continuing this way the whole structure is build. In this way the structure in Figure 2.12 is build, from left to right. Notice that the base of the recursion are singleton lists.

The crucial point that makes this function work correctly is that recursive calls return the next free memory location, thus taking care of the bookkeeping.

Array access

Once an array has been constructed, it can be used to assign values to its entries, and to lookup values, *i.e.* to access the array. A function access_at which is used to translate array access, is defined in the following way:

$$[\![\texttt{a[i]}]\!] \stackrel{\text{def}}{=} \text{access\_at}(\text{get\_typ}, [\![\texttt{a}]\!], [\![\texttt{i}]\!])$$

assuming that `a[i]` is not the left hand side of an assignment. The function get_typ is determined by the component type of the array `a`, for example: if `a` is an integer array of type `int[]`, then get_typ $=$ get_int. And if `a` is a 2-dimensional array of, say Booleans, then get_typ $=$ get_ref.

The JAVA evaluation strategy prescribes that first the array expression, and then the index expression must be evaluated. Subsequently it must be checked first if the array reference is non-null, and then it is checked if the (evaluated) index is non-negative and smaller than the length of the array. Only then the memory can be accessed (see [GJSB00, §§ 15.12.1 and §§ 15.12.2]).

The type-theoretic function access_at makes use of an auxiliary function access_at_aux. This is done only for clarity of the presentation[6]. The function access_at evaluates all the arguments, in the prescribed order, and checks that they all return a normal result.

– TYPE THEORY

$$
\begin{array}{l}
\mathsf{get\_typ} : \mathsf{OM} \times \mathsf{MemAdr} \to \mathsf{typ}, \\
a : \mathsf{OM} \to \mathsf{ExprResult[OM, RefType]}, \\
i : \mathsf{OM} \to \mathsf{ExprResult[OM, int]} \ \vdash \\
\quad \mathsf{access\_at}(\mathsf{get\_typ}, a, i) \ : \ \mathsf{OM} \to \mathsf{ExprResult[OM, Out]} \ \stackrel{\text{def}}{=} \\
\quad\quad \lambda x : \mathsf{OM}.\ \ \mathsf{CASE}\ a\,x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto \\
\quad\quad\quad\quad\quad \mathsf{CASE}\ i\ (y.\mathsf{ns})\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\, z \mapsto \mathsf{access\_at\_aux}\ \ (\mathsf{get\_typ}, y.\mathsf{res}, z.\mathsf{res}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (z.\mathsf{ns}) \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\, c \mapsto \mathsf{abnorm}\, c\ \} \\
\quad\quad\quad\quad \mid \mathsf{abnorm}\, b \mapsto \mathsf{abnorm}\, b\ \}
\end{array}
$$

If evaluation of all the arguments terminates normally, the function access_at_aux is called, which checks whether the reference to the array is a non-null reference and next, whether the index is a value between the array bounds, *i.e.* between 0 and the length of the array. If this is not the case, an `ArrayIndexOutOfBoundsException` is thrown, otherwise the appropriate value is returned.

– TYPE THEORY

$$
\begin{array}{l}
\mathsf{get\_typ} : \mathsf{OM} \times \mathsf{MemAdr} \to \mathsf{typ}, \\
a : \mathsf{RefType}, i : \mathsf{int} \ \vdash \\
\quad \mathsf{access\_at\_aux}(\mathsf{get\_typ}, a, i) \ : \ \mathsf{OM} \to \mathsf{ExprResult[OM, Out]} \ \stackrel{\text{def}}{=} \\
\quad\quad \lambda x : \mathsf{OM}.\ \ \mathsf{CASE}\ a\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad \mid \mathsf{null} \mapsto [\![\texttt{new NullPointerException()}]\!] \\
\quad\quad\quad\quad \mid \mathsf{ref}\, r \mapsto \\
\quad\quad\quad\quad\quad \mathsf{IF}\ i < 0 \vee i \geq (\mathsf{get\_dimlen}\, r\, x).\mathsf{len} \\
\quad\quad\quad\quad\quad \mathsf{THEN}\ [\![\texttt{new ArrayIndexOutOfBoundsException()}]\!] \\
\quad\quad\quad\quad\quad \mathsf{ELSE\ norm}\ \ (\mathsf{ns} = x, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{res} = \mathsf{get\_typ}(\mathsf{heap}(\ \mathsf{ml} = r,\ \ \mathsf{cl} = i\ ))\, x)\}
\end{array}
$$

Accessing values in a multi-dimensional array is translated by using multiple access_at functions. *E.g.* `a[2][3]` is translated as follows.

– TYPE THEORY

$$
\begin{array}{rl}
& [\![\texttt{a[2][3]}]\!] \\
= & [\![\texttt{(a[2])[3]}]\!] \\
= & \mathsf{access\_at}(\mathsf{get\_int}, [\![\texttt{a[2]}]\!], 3) \\
= & \mathsf{access\_at}(\mathsf{get\_int}, \mathsf{access\_at}(\mathsf{get\_ref}, [\![\texttt{a}]\!], 2), 3)
\end{array}
$$

[6]In the semantic description of JAVA in PVS and ISABELLE/HOL, this is simply written as one function

– TYPE THEORY

$a, data$: OM $\to$ ExprResult[OM, RefType], i : OM $\to$ ExprResult[OM, int] $\vdash$

ref_assign_at$(a, i)(data)$: OM $\to$ ExprResult[OM, RefType] $\stackrel{\text{def}}{=}$

λx : OM. CASE $a\,x$ OF {
| hang $\mapsto$ hang
| norm $y \mapsto$ CASE i (y.ns) OF {
| hang $\mapsto$ hang
| norm $z \mapsto$ CASE $data$ (z.ns) OF {
| hang $\mapsto$ hang
| norm $w \mapsto$ ref_assign_at_aux
(y.res, z.res, w.res)(w.ns)
| abnorm $d \mapsto$ abnorm d}
| abnorm $c \mapsto$ abnorm c }
| abnorm $b \mapsto$ abnorm b }

Figure 2.13: Definition of ref_assign_at

Thus, the inner call to access_at returns the array `a[2]`, and in this array, the third entry is returned by the outer call to access_at.

Array assignment

The last operation that is discussed in this section is array assignment. Here a distinction has to be made between assigning primitive values and reference values. For primitive values it can be statically checked (by the compiler) whether the element is storable to the array, but for references this check can only be done at run-time. If an attempt is made to store an unstorable element, an `ArrayStoreException` is thrown. Consider for example the following JAVA program fragment.

– JAVA

```
class A {}

class B1 extends A{}

class B2 extends A{}

class C {
  void m() {
  A [] A_array = new B1 [2];
  A a = new B2();
  A_array[0] = a;
  }
}
```

a : RefType, i : int, $data$: RefType $\vdash$

ref_assign_at_aux$(a, i)(data)$: OM $\to$ ExprResult[OM, RefType] $\stackrel{\text{def}}{=}$

λx : OM.CASE a OF {
| null $\mapsto$ ⟦`new NullPointerException()`⟧
| ref r $\mapsto$
IF $i < 0 \vee i \geq$ (get_dimlen $r\ x$).len
THEN ⟦`new ArrayIndexOutOfBoundsExceptions()`⟧
ELSE
CASE $data$ OF {
| null $\mapsto$ norm (ns = put_ref(heap(cl = r, ml = i))
x $data$,
res = $data$)
| ref d $\mapsto$ IF (get_type $r\ x$ = `"Object"`
$\wedge$
(get_dimlen $r\ x$).dim $\leq$ (get_dimlen $d\ x$).dim)
$\vee$
(SubClass? (get_type $d\ x$) (get_type $r\ x$)
$\wedge$
(get_dimlen $r\ x$).dim = (get_dimlen $d\ x$).dim)
THEN norm(ns = put_ref
(heap(cl = r, ml = i))
x
$data$,
res = $data$)
ELSE ⟦`new ArrayStoreException()`⟧}

Figure 2.14: Definition of the auxiliary function ref_assign_at_aux

This array assignment is accepted by the compiler, since both the elementtype of `A_array` and the variable `a` are declared as subclasses of `A`. However, at run-time, the elementtype of the array is `B1`, while `a` is an instance of class `B2` (which is unrelated to `B1`). Thus, an `ArrayStoreException` will be thrown.

The function ref_assign_at (in Figure 2.13) describes the semantics of assigning references to an array. Again, the definition of ref_assign_at uses an auxiliary function ref_assign_at_aux. The function ref_assign_at evaluates all the arguments of the array assignment in the order prescribed by the JAVA language specification, *i.e.* first the array expression, then the index expression and finally the argument to the assignment (the data expression). If evaluation of all these arguments terminates normally, ref_assign_at_aux(defined in Figure 2.14) is called, which checks (1) if the array is a non-null reference, (2) if the (evaluated) index is between the array bounds, *i.e.* between 0 and the length of the array, and (3) if the data value is storable in the array, *i.e.* for non-null references it is checked whether the run-time element is assignable to the array. This check is basically the same as the one performed by the function CheckCast, as explained in Section 2.6.6.

The function prim_assign_at, describing the semantics of assigning primitive values, is similar, but leaves out the "storability" check. In contrast to the function ref_assign_at_aux, which has to check whether the element is storable, the primitive assignment function can immediately store the element. The language definition guarantees that the element is storable in the array. The function prim_assign_at has an extra parameter put_typ, similar to the get_typ parameter in the function access_at. The actual parameter put_typ can be determined from the static type of the array.

2.6 Classes, objects and inheritance

What has been discussed so far, describes a semantics for the imperative part of JAVA, which does not include object-oriented features, such as inheritance, overriding of methods, dynamic method lookup and hiding of fields. This section will describe a semantics for these object-oriented concepts. It is tailored towards JAVA, but the ideas could be adapted to describe the semantics of other object-oriented programming languages as well.

The semantics that is presented in this section gives rise to a large number of different definitions for each concrete class. Later, in Chapter 4, a compiler is described which performs the translation from JAVA classes to definitions automatically (generating definitions in the input languages for the theorem provers PVS and ISABELLE). Therefore, it is important to keep in mind that all definitions presented below are generated automatically, and do not have to be given by hand.

Recall from Section 1.1 that a JAVA class consists of the following ingredients: a name, a superclass, super interfaces, fields, methods and constructors. Together, but without the method and constructor bodies, they describe the interface or signature of a class. Declarations of fields, methods and constructors can be preceded by modifiers, such as `public`, `private`, `static` and `final`, but we abstract away from these. In some cases these modifiers require small changes in the translation, but they do not affect the general ideas.

For each concrete class, a semantics can be given in terms of coalgebras. Here coalgebras are only used to conveniently combine all the ingredients of a class in a single function. Specifically,

n functions $f_1 : \mathsf{Self} \to \sigma_1, \ldots, f_n : \mathsf{Self} \to \sigma_n$ with a common domain can be combined in one function $\mathsf{Self} \to [\, f_1 : \sigma_1, \ldots, f_n : \sigma_n \,]$ with a labeled product type as codomain[7], forming a coalgebra. As discussed in Chapter 1 coalgebras give rise to a general theory of behaviour for dynamic systems, involving useful notions like invariance and bisimilarity. In our semantics the use of coalgebras remains fairly superficial. However, it is important to realise that classes are modelled as coalgebras, because this immediately allows us apply the theory of coalgebras on our formalisation, resulting in many interesting possibilities to extend the work presented here. For more background information, see [JR97].

The translation of a JAVA class consists of two parts. First, a semantic description of the interface of the class is given. Next, the fields are bound to actual memory locations and methods are bound to method bodies.

The translation of JAVA interfaces follows closely the translation of the interfaces of JAVA classes. Naturally, the second part of the translation, where method names are bound to method bodies is not relevant for JAVA interfaces. Here we will not go into the differences of the semantics of JAVA classes and JAVA interfaces.

2.6.1 A single class

First JAVA classes are considered in isolation, without looking at the inheritance structure. The semantics of each class is described using a single coalgebra. The easiest way to understand the translation from classes to coalgebras is by looking at an example. Suppose we have the following JAVA class.

– JAVA

```
class MyClass {

  int i;
  int k = 3;

  void m (byte a, int b) {  // i becomes max(a, b)
    if (a > b) {
         i = a;
       }
    else i = b;
  }

  MyClass() {
    i = 6;
  }
}
```

The class `MyClass` contains two fields, `i` and `k`, and one method `m`. Furthermore, this class contains a constructor `MyClass()`, which creates a new object in `MyClass`, initialises all its fields, either to the explicitly stated values (thus `k` is set to 3), or to their default values (`i` is

[7] Alternatively, one can combine these n functions into elements of a so-called "trait type" $[\, f_1 : \mathsf{Self} \to \sigma_1, \ldots, f_n : \mathsf{Self} \to \sigma_n \,]$, like in [AC96, §§8.5.2].

set to 0), and subsequently executes its body, where `i` is set to 6. Constructors are often left implicit. In that case, their only effect is to initialise the fields of a new object to its default values. Constructors can be distinguished from normal methods by the following: they have the same name as the class, and no return type (nor `void`) is given explicitly. Constructors in JAVA are called immediately after a `new` expression, which return a reference to the newly created object. Notice that, since constructors also perform certain initialisations, they are really state transformers.

The class `MyClass` gives rise to a definition of a labeled product type MyClassIFace in type theory.

– TYPE THEORY

$$\begin{aligned}&\mathsf{MyClassIFace[Self]} : \mathsf{TYPE} \stackrel{\text{def}}{=} \\ &\quad [\ldots \ \text{// For the superclass, see Section 2.6.2} \\ &\quad\ \mathsf{i} : \mathsf{int}, \\ &\quad\ \mathsf{i\_becomes} : \mathsf{int} \to \mathsf{Self}, \\ &\quad\ \mathsf{k} : \mathsf{int}, \\ &\quad\ \mathsf{k\_becomes} : \mathsf{int} \to \mathsf{Self}, \\ &\quad\ \mathsf{m\_byte\_int} : \mathsf{byte} \to \mathsf{int} \to \mathsf{StatResult[Self]}, \\ &\quad\ \mathsf{constr\_MyClass} : \mathsf{ExprResult[Self, RefType]}\ ]\end{aligned}$$

There are several things worth noticing here.

- The field declaration `int i` gives rise not only to a label i: int (= ⟦`int`⟧) in the product type, which is used for field access, but also to an associated assignment operation, with label i_becomes. This assignment operation takes an integer as input, and produces a new state in Self, in which the state is changed in such a way that the i field is changed to the argument of the assignment operation (and the rest is unchanged). Similarly for `k`. Variable initialisers (like `k = 3`) are ignored at this stage, since they are irrelevant for the interface type (just like method bodies).

- The method `m` which is a void method, is modeled as a field of the labeled product of type StatResult[Self]. Its name `m` is extended with types of its arguments, resulting in a label m_byte_int. This is done to avoid identical labels within the product type. In JAVA it is allowed to have two methods with the same name in one class, as long as they can be distinguished by the types and number of their arguments. Thus, by adding this information to the label name, identical label names are avoided[8]. Similarly, methods with a return value are modeled as expressions, *e.g.* `int n() {return 3;}` would give rise to a field n with type ExprResult[Self, int]

- The translation of the constructor `MyClass` is prefixed with a tag constr_, thus avoiding possible name clashes. If the class would have constructors with arguments, these names would also have been extended with the types of the arguments, similar to the extension

[8]The translation from JAVA program code to PVS or ISABELLE theories includes even more precautions: special symbols (`?` in PVS and `'` in ISABELLE, respectively) which are not allowed in JAVA identifiers are added to the generated names, thus avoiding name clashes between *e.g.* a method `m` with a parameter of type `byte` and a field with name `m_byte`. For more information, see Section 4.2.

of the name of method `m`. The type of the constructor is implicit in the JAVA code, but has to be made explicit in the type theoretic formalisation. Since a constructor returns a reference to a newly created object, it is modeled as a field with type ExprResult[Self, RefType]. More detailed information on constructors and the typical aspects of their semantics is given in Section 2.6.11.

Possible `throws` clauses [GJSB00, §§8.4.4] in method (or constructor) declarations – indicating which (explicit) exceptions can be thrown by the method – are ignored throughout the translation. From the language definition follows that `throws` clauses are always given if necessary, and for our translation we assume that the code is accepted by the JAVA compiler. These clauses play no role in the type theoretic semantics.

The types occurring in the above interface type MyClassIFace describe the "visible" signatures of the fields, methods and constructors in the JAVA class `MyClass`. But in object-oriented programming there is always an invisible argument to a field/method/constructor, namely the current state in which the field/ method/constructor is invoked. This is made explicit by modelling classes as *coalgebras* for interface types, *i.e.* as functions of the form:

$$\mathsf{Self} \xrightarrow{\quad c \quad} \mathsf{MyClassIFace[Self]}$$

Such a coalgebra actually combines the fields, methods and constructors of the class in a single function. These are made explicit, using the isomorphism $\mathsf{Self} \to [\, f_1 : \sigma_1, \ldots, f_n : \sigma_n \,] \cong [\, f_1 : \mathsf{Self} \to \sigma_1, \ldots, f_n : \mathsf{Self} \to \sigma_n \,]$, via what we call "extraction" functions:

– TYPE THEORY –

$$c : \mathsf{Self} \to \mathsf{MyClassIFace[Self]} \vdash$$
$$\mathsf{i}(c) \;:\; \mathsf{Self} \to \mathsf{int} \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{Self}.\ (c\, x).\mathsf{i}$$

$$c : \mathsf{Self} \to \mathsf{MyClassIFace[Self]} \vdash$$
$$\mathsf{i\_becomes}(c) \;:\; \mathsf{Self} \to \mathsf{int} \to \mathsf{Self} \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{Self}.\ ((c\, x).\mathsf{i\_becomes})$$

$$c : \mathsf{Self} \to \mathsf{MyClassIFace[Self]} \vdash$$
$$\mathsf{k}(c) \;:\; \mathsf{Self} \to \mathsf{int} \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{Self}.\ (c\, x).\mathsf{k}$$

$$c : \mathsf{Self} \to \mathsf{MyClassIFace[Self]} \vdash$$
$$\mathsf{k\_becomes}(c) \;:\; \mathsf{Self} \to \mathsf{int} \to \mathsf{Self} \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{Self}.\ ((c\, x).\mathsf{k\_becomes})$$

$$a : \mathsf{byte}, b : \mathsf{int}, c : \mathsf{Self} \to \mathsf{MyClassIFace[Self]} \vdash$$
$$\mathsf{m\_byte\_int}(a)(b)(c) \;:\; \mathsf{Self} \to \mathsf{StatResult[Self]} \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{Self}.\ ((c\, x).\mathsf{m\_byte\_int})(a)(b)$$

$$c\colon \mathsf{Self} \to \mathsf{MyClassIFace[Self]} \vdash$$
$$\mathsf{constr\_MyClass}(c)\ :\ \mathsf{Self} \to \mathsf{ExprResult[Self, RefType]} \stackrel{\text{def}}{=}$$
$$\lambda x\colon \mathsf{Self}.\ ((c\ x).\mathsf{constr\_MyClass})$$

The coalgebra $c\colon \mathsf{Self} \to \mathsf{MyClassIFace[Self]}$ above thus combines all the operations of the class `MyClass`. In the remainder of this text, we shall always describe operations – fields (with their assignments), methods and constructors – of a class, say `A`, using extraction definitions as above, applied to a coalgebra of type AIFace.

2.6.2 Inheritance and nested interface types

In JAVA every class (except `Object`) inherits from exactly one other class, either explicitly, denoted by the `extends` keyword, or implicitly from `Object`. Thus, to model JAVA classes faithfully in our type theory, we have to take inheritance into account. Again, we look at an example. Suppose we have the following JAVA class, inheriting from `MyClass` described above.

– JAVA

```
class MySubClass extends MyClass {

  int j;

  int n (byte a) {
    m(a, 3);
    return i;
  }
}
```

The new class `MySubClass` inherits the field `i` and method `m` of `MyClass`, and it declares its own field `j` and method `n`. As can be seen in the body of the method `n`, the methods and fields from the superclass are immediately available, *i.e.* the method `m` and field `i` are called without any visible further reference to `MyClass`; it uses the implicit self reference to the current object (the `this` reference). This should also be possible in our semantics.

This class gives rise to the following interface type in type theory.

– TYPE THEORY

$$\mathsf{MySubClassIFace[Self]}\ :\ \mathsf{TYPE} \stackrel{\text{def}}{=}$$
$$[\,\mathsf{super\_MyClass}\colon \mathsf{MyClassIFace[Self]},$$
$$\mathsf{j}\colon \mathsf{int},$$
$$\mathsf{j\_becomes}\colon \mathsf{int} \to \mathsf{Self},$$
$$\mathsf{n\_byte}\colon \mathsf{byte} \to \mathsf{ExprResult[Self, int]},$$
$$\mathsf{constr\_MySubClass}\colon \mathsf{ExprResult[Self, RefType]}\,]$$

Comparing this labeled product MySubClassIFace with the labeled product type MyClassIFace,

the important difference is the occurrence of a label super_MyClass (with type MyClassIFace). This is the formalisation of the inheritance relation between `MySubClass` and `MyClass`.

Thus, via this link, the methods and fields of `MyClass` are available. In a similar way, MyClassIFace[Self] contains a field super_Object: ObjectIFace[Self], formalising the implicit inheritance from `Object` by `MyClass`. The labeled product ObjectIFace is in fact the only interface type (generated from a JAVA class definition), which does not contain a super field.

Notice that the constructor of `MySubClass`, which is implicit in the JAVA code, is made explicit in the interface type.

Just as for `MyClass`, we get a coalgebra for `MySubClass`, capturing its methods and fields.

– TYPE THEORY

$$\mathsf{Self} \xrightarrow{\quad c \quad} \mathsf{MySubClassIFace[Self]}$$

Again, we define appropriate extraction functions for its methods and fields. To access the fields and methods in `MyClass`, an extraction function super_MyClass is defined. It transforms MySubClassIFace coalgebras into MyClassIFace coalgebras. Later in Section 2.6.4 we shall see another way to perform this transformation, needed for casting.

– TYPE THEORY

$$c\colon \mathsf{Self} \to \mathsf{MySubClassIFace[Self]} \vdash$$
$$\mathsf{super\_MyClass}(c)\ :\ \mathsf{Self} \to \mathsf{MyClassIFace[Self]} \stackrel{\text{def}}{=}$$
$$\lambda x\colon \mathsf{Self}.\ ((c\,x).\mathsf{super\_MyClass})$$

However, to be able to access the methods and fields from `MyClass` immediately (as can be done in JAVA), this is not enough. Therefore, we also define immediate extraction functions for the methods and fields of `MyClass`, working on the coalgebra for `MySubClass`. Thus we get the following definitions (among others).

– TYPE THEORY

$$c\colon \mathsf{Self} \to \mathsf{MySubClassIFace[Self]} \vdash$$
$$\mathsf{i}(c)\ :\ \mathsf{Self} \to \mathsf{int} \stackrel{\text{def}}{=}$$
$$\mathsf{i}(\mathsf{super\_MyClass}(c))$$

$$c\colon \mathsf{Self} \to \mathsf{MySubClassIFace[Self]} \vdash$$
$$\mathsf{i\_becomes}(c)\ :\ \mathsf{Self} \to \mathsf{int} \to \mathsf{Self} \stackrel{\text{def}}{=}$$
$$\mathsf{i\_becomes}(\mathsf{super\_MyClass}(c))$$

$$a\colon \mathsf{byte}, b\colon \mathsf{int}, c\colon \mathsf{Self} \to \mathsf{MySubClassIFace[Self]} \vdash$$
$$\mathsf{m\_byte\_int}(a)(b)(c)\ :\ \mathsf{Self} \to \mathsf{StatResult[Self]} \stackrel{\text{def}}{=}$$
$$\mathsf{m\_byte\_int}(a)(b)(\mathsf{super\_MyClass}(c))$$

Note how this involves overloading, because for instance the extraction function $\mathsf{i}(c)$ is defined both for coalgebras of type Self $\to$ MyClassIFace[Self] and for coalgebras of type Self $\to$ MySubClassIFace[Self], representing the classes `MyClass` and `MySubClass`, respectively.

For convenience, also the following abbreviations are defined.

– TYPE THEORY

$$c : \mathsf{Self} \to \mathsf{MySubClassIFace[Self]} \vdash$$
$$\mathsf{MySubClass\_sup\_MyClass}(c) \;:\; \mathsf{Self} \to \mathsf{MyClassIFace[Self]} \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{super\_MyClass}(c\,x).$$

$$c : \mathsf{Self} \to \mathsf{MySubClassIFace[Self]} \vdash$$
$$\mathsf{MySubClass\_sup\_Object}(c) \;:\; \mathsf{Self} \to \mathsf{ObjectIFace[Self]} \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{super\_Object}(\mathsf{super\_MyClass}(c\,x)).$$

2.6.3 Invariants

From the types of fields, methods and constructors we get an immediate definition for class invariants [HJ98], based on the types of the fields, methods and constructors only. Basically, a property is called a class invariant if it is established by all normally terminating constructors and preserved by all terminating (public) methods. Notice that we require that class invariants are preserved by both normally and abruptly terminating methods. As the compiler ensures that return, break and continue abnormalities are caught within a method, the only cause for abrupt termination that has to be considered *w.r.t* class invariants are exceptions. More precisely, a predicate P : Self $\to$ bool is a class invariant for a class C, if it satisfies the following conditions.

1. For each constructor c in class C, if c terminates normally, resulting in a state x, then the predicate P should be true for this state x.

2. For each method m in C, if it is executed in a state x where $P\,x$ is true, and execution of this method terminates normally or abruptly, resulting in a state y, then also $P\,y$ should hold.

Note that even when a method terminates abruptly, the invariants should hold. This implies that if something goes wrong, a method must throw an exception before any crucial data is corrupted. A consequence is that if the exception is caught at some later stage, the invariant still holds.

For each class, a definition of invariant can be given. For example, for class `MySubClass`, we get the following definitions (using auxiliary functions initially and MySubClassPred) assuming that we have appropriate definitions for class `MyClass` – and recursively for class `Object`.

$$P : \mathsf{Self} \to \mathsf{bool}, c : \mathsf{Self} \to \mathsf{MySubClassIFace[Self]} \vdash$$
$$\mathsf{initially}(P)(c) \ : \ \mathsf{bool} \stackrel{\mathrm{def}}{=}$$
$$\forall x : \mathsf{Self}.\ \ \mathsf{CASE\ constr\_MySubClass}(c)\, x\ \mathsf{OF}\ \{$$
$$\mid \mathsf{hang} \mapsto \mathsf{true}$$
$$\mid \mathsf{norm}\, y \mapsto P(y.\mathsf{ns})$$
$$\mid \mathsf{abnorm}\, a \mapsto \mathsf{true}\ \}$$

$$P : \mathsf{Self} \to \mathsf{bool}, c : \mathsf{Self} \to \mathsf{MySubClassIFace[Self]} \vdash$$
$$\mathsf{MySubClassPred}(P)(c) \ : \ \mathsf{bool} \stackrel{\mathrm{def}}{=}$$
$$\lambda x : \mathsf{Self}.\ \ \mathsf{MyClassPred}(P)(c)\, x \wedge$$
$$\mathsf{CASE\ n\_byte}(c)\, x\ \mathsf{OF}\ \{$$
$$\mid \mathsf{hang} \mapsto \mathsf{true}$$
$$\mid \mathsf{norm}\, y \mapsto P(y.\mathsf{ns})$$
$$\mid \mathsf{abnorm}\, a \mapsto \ \mathsf{CASE}\ a\ \mathsf{OF}\ \{$$
$$\mid \mathsf{excp}\, e \mapsto P(e.\mathsf{es})$$
$$\mid \mathsf{rtrn}\, r \mapsto \mathsf{true}$$
$$\mid \mathsf{break}\, b \mapsto \mathsf{true}$$
$$\mid \mathsf{cont}\, c \mapsto \mathsf{true}\ \}\}$$

$$P : \mathsf{Self} \to \mathsf{bool}, c : \mathsf{Self} \to \mathsf{MySubClassIFace[Self]} \vdash$$
$$\mathsf{invariant}(P)(c) \ : \ \mathsf{bool} \stackrel{\mathrm{def}}{=}$$
$$\mathsf{initially}(P)(c) \wedge$$
$$\forall x : \mathsf{Self}.\ P\, x \supset \mathsf{MySubClassPred}(P)(c)\, x$$

An example verification of a class invariant property for JAVA's `Vector` class is discussed in Section 7.1.

2.6.4 Overriding and hiding

So far, we have only seen an example of inheritance where the subclass `MySubClass` simply adds extra fields and methods to the superclass. But the same field and method names may also be reappear in subclasses. In JAVA this is called *hiding* of fields, and *overriding* of methods. The possibility to override a method in a subclass allows a programmer to give a new implementation for a method in a subclass[9]. Which implementation is actually used, depends on the run-time type of the object on which the method is called. Hiding of fields occurs if a subclass contains a field with the same name as a field in one of its superclasses. From methods in this subclass, the field in the superclass can only be accessed by explicitly using `super` or another reference

[9]Preferably, this new implementation does not change the observable behaviour of the method *w.r.t.* the superclass, *i.e.* it is a behavioural subtype of the original method [LW94]. However, to be able to reason about arbitrary JAVA programs, nothing is assumed about the new implementation here.

of your superclass's type. However, if a method in the superclass is executed, it uses the field from the superclass (since the binding of fields is based on the static type)[10].

Notice that, with these mechanisms, field selection is based on the static type of the receiving object, whereas method selection is based on the dynamic (or run-time) type of an object. The latter mechanism is often referred to as dynamic method lookup, or late binding. Consider the following example.

– JAVA

```
class A {
  int i = 1;
  int m() { return i * 100; }
}
class B extends A {
  int i = 10;
  int m() { return i * 1000; }
}
class Test {
  int test1() {
    A[] ar = { new A(), new B() };
    return ar[0].i + ar[0].m() + ar[1].i + ar[1].m();
  }
}
```

The field `i` in the subclass `B` hides the field `i` in the superclass `A`, and similarly, the method `m` in `B` overrides the method `m` in `A`. In the `test1` method of class `Test` a local variable `ar` of type 'array of `A`s' is declared and initialised with length 2 containing a new `A` object at position 0, and a new `B` object at position 1. Note that at position 1 there is an implicit conversion from `B` to `A` to make the new `B` object fit into the array of `A`s. Interestingly, the `test1` method will return `ar[0].i + ar[0].m() + ar[1].i + ar[1].m()`, which is `1 + 1 * 100 + 1 + 10 * 1000 = 10102`, because: when `new B()` is converted to type `A` the hidden field becomes visible again, so the field `ar[1].i` refers to `i` in `A`, but the overriding method replaces the original method, thus the method `ar[1].m()` leads to execution of `m` in `B` (which uses the field `i` from `B`). See [AG97, §§3.4], or also [GJSB00, §§8.4.6.1]:

> Note that a qualified name or a cast to a superclass is not effective in attempting to access an overridden method; in this respect, overriding of methods differs from hiding of fields.

It is a challenge to provide a semantics for this behaviour. We do so by using a special cast function between coalgebras, which performs appropriate replacements of methods and fields. To explain this, another example is discussed, in which the inheritance structure of `MyClass` and `MySubClass` is extended with another subclass: class `AnotherSubClass`.

[10] Hiding of fields is allowed in JAVA in order to allow implementors of existing superclasses to add new fields without breaking subclasses [AG97].

– JAVA

```
class AnotherSubClass extends MySubClass {
                        // recall MySubClass
                        // extends MyClass
  int i; // hides i from MyClass

  // overrides m from MyClass
  void m (byte a, int b) {
    if (a <  b) {
         i = a;
       }
    else i = b;
  }
}
```

Again, we get an interface AnotherSubClassIFace, capturing the fields, methods, constructors and the superclass of this class, and corresponding extraction functions. Notice that AnotherSubClassIFace contains m and i twice: once directly, and once inside the nested interface type MyClassIFace. Thus two extraction functions are defined for each of them.

– TYPE THEORY

$$c : \mathsf{Self} \to \mathsf{AnotherSubClassIFace[Self]} \vdash$$
$$\mathsf{i}(c) \ : \ \mathsf{Self} \to \mathsf{int} \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{Self}.\ (c\,x).\mathsf{i}$$

$$c : \mathsf{Self} \to \mathsf{AnotherSubClassIFace[Self]} \vdash$$
$$\mathsf{MyClass\_i}(c) \ : \ \mathsf{Self} \to \mathsf{int} \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{Self}.\ \mathsf{i}(\mathsf{AnotherSubClass\_sup\_MyClass}(c))$$

$$a : \mathsf{byte}, b : \mathsf{int}, c : \mathsf{Self} \to \mathsf{AnotherSubClassIFace[Self]} \vdash$$
$$\mathsf{m\_byte\_int}(a)(b)(c) \ : \ \mathsf{Self} \to \mathsf{StatResult[Self]} \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{Self}.\ ((c\,x).\mathsf{m\_byte\_int})(a)(b)$$

$$a : \mathsf{byte}, b : \mathsf{int}, c : \mathsf{Self} \to \mathsf{AnotherSubClassIFace[Self]} \vdash$$
$$\mathsf{MyClass\_m\_byte\_int}(a)(b)(c) \ : \ \mathsf{Self} \to \mathsf{StatResult[Self]} \stackrel{\text{def}}{=}$$
$$\mathsf{m\_byte\_int}(a)(b)(\mathsf{AnotherSubClass\_sup\_MyClass}(c))$$

The extraction functions MyClass_i and MyClass_m_byte_int are used to translate calls to `super.i` and `super.m()`.

What is needed to describe the behaviour of this class is a semantics of "casting", *i.e.* a way to denote a cast from an AnotherSubClass coalgebra $c : \mathsf{Self} \to \mathsf{AnotherSubClassIFace[Self]}$

to a MyClass coalgebra AnotherSubClass2MyClass(c) : Self $\to$ MyClassIFace[Self] which incorporates the differences between hiding and overriding. Just taking the super_MyClass entry (via the super_MySubClass) is not good enough: we need additional updates, which select the fields of the superclass `MyClass`, but the methods of the subclass `AnotherSubClass`.

Therefore, we define cast operations as functions which transform coalgebras (representing objects) to coalgebras of the superclass, with appropriate bindings of methods and fields. As an example, we look at the cast operations from `AnotherSubClass` to its superclasses.

– TYPE THEORY

$$c : \mathsf{Self} \to \mathsf{AnotherSubClassIFace[Self]} \vdash$$
$$\mathsf{AnotherSubClass2Object}(c) \;:\; \mathsf{Self} \to \mathsf{ObjectIFace[Self]} \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{Self}.\, \mathsf{AnotherSubClass\_sup\_Object}(c)\, x$$

$$c : \mathsf{Self} \to \mathsf{AnotherSubClassIFace[Self]} \vdash$$
$$\mathsf{AnotherSubClass2MyClass}(c) \;:\; \mathsf{Self} \to \mathsf{MyClassIFace[Self]} \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{Self}.\;\; \mathsf{AnotherSubClass\_sup\_MyClass}(c)\, x \;\mathsf{WITH}$$
$$(\, \mathsf{super\_Object} = \mathsf{AnotherSubClass2Object}(c)\, x,$$
$$\mathsf{m\_byte\_int} = \lambda a : \mathsf{byte}.\, \lambda b : \mathsf{int}.\, \mathsf{m\_byte\_int}(a)(b)(c)\, x\,)$$

$$c : \mathsf{Self} \to \mathsf{AnotherSubClassIFace[Self]} \vdash$$
$$\mathsf{AnotherSubClass2MySubClass}(c) \;:\; \mathsf{Self} \to \mathsf{MySubClassIFace[Self]} \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{Self}.\;\; \mathsf{AnotherSubClass\_sup\_MySubClass}(c)\, x \;\mathsf{WITH}$$
$$(\, \mathsf{super\_MyClass} = \mathsf{AnotherSubClass2MyClass}(c)\, x\,)$$

The coalgebras that are returned by these cast operations model "run-time" tables for field and method lookup, returning the fields and methods that are in the scope of the object.

The crucial thing to notice is that, if a cast takes place from `AnotherSubClass` to `MyClass`, this returns a labeled product in which the label m_byte_int is still bound to m_byte_int from AnotherSubClassIFace. Thus:

$$\mathsf{m\_byte\_int}(\mathsf{AnotherSubClass2MyClass}(c)) = \mathsf{m\_byte\_int}(c)$$

In contrast, the label i is bound to the label i from MyClassIFace, thus:

$$\mathsf{i}(\mathsf{AnotherSubClass2MyClass}(c)) = \mathsf{MyClass\_i}(c)$$

Thus, the casting results in a coalgebra which has the static type of the superclass, but provides the dynamic behaviour of the subclass.

In general, all overriding methods from a subclass replace the methods from its superclass. Hidden fields reappear in such casting because they are not replaced. Below, in Section 2.6.9, it is discussed how method bodies are called with appropriately cast coalgebras.

2.6.5 Extending the extraction functions

The extraction functions for methods (or constructors) with arguments described above cannot be used immediately. They are defined in such a way that their formal parameters are values. But, in a method call, the actual parameters might be complicated expressions, which first have to be evaluated (and might throw exceptions or not terminate at all). These arguments thus should be modeled as expressions in JAVA. The evaluation order of JAVA prescribes that first the arguments are evaluated, then the method lookup is done and finally the method body is executed [GJSB00, §§15.11.4]. In our semantics this is modeled with method extension functions, which get expressions as arguments (instead of values). For every method or constructor with arguments, a method extension function is defined. A monadic description of extension functions is described in [JP00b]. Method extension functions first evaluate the arguments of a method (from left to right), and then call the appropriate extraction function. Notice that if a method does not have arguments, it is not necessary to define a method extension function for it, since the extraction function can be used immediately. An example of a method extension function is the method extension function for method `n` in `MySubClass`. Notice the overloading with the extraction function n_byte for MySubClassIFace. This does not cause any problems, because the types of the arguments are different (byte versus Self $\to$ ExprResult[Self, byte]).

– TYPE THEORY

$$a : \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{byte}],\ c : \mathsf{Self} \to \mathsf{MySubClassIFace}[\mathsf{Self}] \vdash$$

$$\mathsf{n\_byte}(a)(c) \ :\ \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{int}] \stackrel{\text{def}}{=}$$

$$\lambda x : \mathsf{Self}.\ \ \mathsf{CASE}\ a\,x\ \mathsf{OF}\ \ \{\mathsf{hang} \mapsto \mathsf{hang}$$
$$\mid \mathsf{norm}\, y \mapsto \mathsf{n\_byte}(y.\mathsf{res})(c)(y.\mathsf{ns})$$
$$\mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a\ \}$$

For inherited methods, the method extension function from the super class is used, working on a "cast coalgebra", thus possible overridings are preserved.

– TYPE THEORY

$$a : \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{byte}],$$
$$b : \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{int}],$$
$$c : \mathsf{Self} \to \mathsf{MySubClassIFace}[\mathsf{Self}] \vdash$$

$$\mathsf{m\_byte\_int}(a)(b)(c) \ :\ \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}] \stackrel{\text{def}}{=}$$
$$\mathsf{m\_byte\_int}(a)(b)(\mathsf{MySubClass2MyClass}(c))$$

Also the extraction functions for field lookup and field assignment are not immediately usable. A field lookup in JAVA is an expression, thus it should be translated into a state transformer Self $\to$ ExprResult[Self, Out] for the appropriate result type Out. However, the extraction functions for fields have type Self $\to$ Out. To bridge this gap, a function F2E (for field-to-expression) is defined, and every field lookup is wrapped-up by this function, so that it becomes an expression.

– TYPE THEORY

$$var : \mathsf{Self} \to \mathsf{Out} \vdash$$

$$\mathsf{F2E}(var) \ :\ \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{Out}] \stackrel{\text{def}}{=}$$
$$\lambda x : \mathsf{Self}.\, \mathsf{norm}(\mathsf{ns} = x, \mathsf{res} = var\, x)$$

A similar approach is taken to wrap-up assignments, so that they become expressions[11]. However, a little bit more work is required, since assignments have an argument, namely the value that has to be assigned. Thus, just like for methods with arguments, an extension is needed in which the argument is evaluated first, before the actual assignment takes place. However, since the number of arguments of the assignment is known (namely 1), this easily can be done within the wrapping function. The wrapping function for assignments, A2E (for assignment-to-expression) is defined as follows.

– TYPE THEORY

$$
\begin{array}{l}
\mathit{var\_becomes}: \mathsf{Self} \to \mathsf{Out} \to \mathsf{Self}, \\
e: \mathsf{Self} \to \mathsf{ExprResult[Self, Out]} \vdash \\
\quad \mathsf{A2E}(\mathit{var\_becomes})(e) \ : \ \mathsf{Self} \to \mathsf{ExprResult[Self, Out]} \stackrel{\text{def}}{=} \\
\quad\quad \lambda x : \mathsf{Self}.\ \ \mathsf{CASE}\ e\ x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad | \ \mathsf{hang} \mapsto \mathsf{hang} \\
\quad\quad\quad\quad\quad\quad | \ \mathsf{norm}\, y \mapsto \mathsf{norm}\ \ (\mathsf{ns} = \mathit{var\_becomes}\ (y.\mathsf{ns})\ (y.\mathsf{res}), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{res} = y.\mathsf{res}\ ) \\
\quad\quad\quad\quad\quad\quad | \ \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a\ \}
\end{array}
$$

2.6.6 The Subclass relation

Based on the inheritance hierarchy of the classes under consideration, a subclass relation can be defined (see [GJSB00, §§8.1.3]). Therefore we define the (reflexive, anti-symmetric and transitive) relation SubClass?: string → string → bool. Remember that, if a class D extends a class C, class D is called a *direct subclass* of C. The SubClass? relationship is the reflexive, transitive closure of this direct subclass relationship[12]. In our semantics it is defined on strings, representing the names of classes. SubClass?("A")("B") is true, when A is a (possibly direct) subclass of B.

Above, in the type-theoretic definition of array assignment (page 44) the subclass relation is already used to check if an element is storable in an array. Also when a reference value is assigned to another reference, a check should be performed which checks if the element is storable, *i.e.* if it can be cast to the other reference, otherwise a run-time exception will be thrown.

Casting is used to enable static typechecking. For example, suppose there is a program fragment with a variable `y` declared as belonging to `Object`. At some point, `y` is known to contain an object in some class `A`, and this value should be assigned to a variable `x` of class `A`. This is done by the following assignment: `A x = (A) y`. Statically it is checked that `y` could possibly contain an object in `A`, because `A` is a subclass of `Object`. At run-time, before the assignment is performed, it is checked that `y` is actually an instance of `A`, otherwise a `ClassCastException` is thrown.

[11] Assignments are modelled as expressions, to allow the translation of JAVA code like *e.g.* `x = (y = 3);`. Remember that expressions can be changed to statements by using the function E2S.

[12] The anti-symmetry of this relation is ensured by the well-formedness of the class hierarchy, which is enforced by the compiler

The function which performs this check, CheckCast, is defined below. It tests whether the cast is allowed, and if so, returns the original reference, otherwise a `ClassCastException` is thrown. This function is defined over the memory model OM, as described in Section 2.5. Thus far, the semantics of classes has been described over some arbitrary state space Self, about which nothing is known *a priori*, but for the CheckCast function it is necessary that the run-time type of objects can be determined, using the functions get_type and get_dimlen. Most other functions below are also defined in terms of get- and put-operations on memory, over the type OM.

– TYPE THEORY

$$
\begin{array}{l}
str : \mathsf{string}, dim : \mathsf{nat}, \\
r : \mathsf{OM} \to \mathsf{ExprResult[OM, RefType]} \vdash \\
\quad \mathsf{CheckCast}(str)(dim)(r) \ : \ \mathsf{OM} \to \mathsf{ExprResult[OM, RefType]} \stackrel{\text{def}}{=} \\
\quad\quad \lambda x : \mathsf{OM}.\ \ \mathsf{CASE}\ r\ x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\quad\quad\quad\quad \mid \mathsf{norm}\ y \mapsto \\
\quad\quad\quad\quad\quad \mathsf{CASE}\ y.\mathsf{res}\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{null} \mapsto r\ x \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{ref}\ p \mapsto \\
\quad\quad\quad\quad\quad\quad\quad \mathsf{IF}\ \ (str = \texttt{"Object"} \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad dim \leq (\mathsf{get\_dimlen}\ p\ (y.\mathsf{ns})).\mathsf{dim}) \\
\quad\quad\quad\quad\quad\quad\quad\quad \vee \\
\quad\quad\quad\quad\quad\quad\quad\quad (\mathsf{SubClass?}\ (\mathsf{get\_type}\ p\ (y.\mathsf{ns}))\ str \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad dim = (\mathsf{get\_dimlen}\ p\ (y.\mathsf{ns})).\mathsf{dim}) \\
\quad\quad\quad\quad\quad\quad\quad \mathsf{THEN}\ r\ x \\
\quad\quad\quad\quad\quad\quad\quad \mathsf{ELSE}\ [\![\texttt{new ClassCastException()}]\!] \\
\quad\quad\quad\quad \mid \mathsf{abnorm}\ a \mapsto \mathsf{abnorm}\ a\ \}
\end{array}
$$

The arguments *str* and *dim* represent the class name and possible dimension that the expression *r* is cast to. If the dimension is 0, this denotes a reference to an object, otherwise it is a reference to an array. For example, we get the following translations.

$$
\begin{array}{c}
[\![\texttt{(A) b}]\!] = \mathsf{CheckCast}\ \texttt{"A"}\ 0\ [\![\texttt{b}]\!] \\
[\![\texttt{(Object [])\ e}]\!] = \mathsf{CheckCast}\ \texttt{"Object"}\ 1\ [\![\texttt{e}]\!]
\end{array}
$$

A distinction is made between casting to `Object` and to other references. An array (with arbitrary dimensions) can be cast to an Object. Thus, in particular a two-dimensional array of Objects can be cast into an one-dimensional array of Objects. For other classes, the dimensions have to be equal (thus a class reference (with dimension 0) can be cast to another class reference, and an n-dimensional array can be cast to another n-dimensional array as long as the run-time type or element type of the cast expression is a subclass of the cast "target". Notice that in the case that *str* is `"Object"`, we left out the subclass-check, because it is trivially satisfied.

2.6.7 Storing fields in memory

At this stage actual cell locations can be connected to the fields of a class. These cell locations are assigned automatically by the LOOP compiler. As explained above, each instance of a class

is stored at some location p : MemLoc in the memory model. The memory cell at this memory location represents the object and thus contains the values of its fields.

For example, the field `i` in class `MyClass` is bound to the first cell locations (0) in the list ints in the memory cell which contains the contents of an object in class `MyClass`. Similarly, `k` is bound to the second cell location (1) in the list ints in this memory cell. This binding is laid down in the following predicates, relating the fields with cell locations.

– TYPE THEORY

$$p\text{: MemLoc}, c\text{: OM} \to \text{MyClassIFace[OM]} \vdash$$
$$\text{i\_cell\_location}(p)(c) \ : \ \text{OM} \to \text{bool} \stackrel{\text{def}}{=}$$
$$\lambda x : \text{OM}.\ \text{i}(c)\, x = \text{get\_int}(\text{heap}(\text{ml} = p, \text{cl} = 0))\, x$$

$$p\text{: MemLoc}, c\text{: OM} \to \text{MyClassIFace[OM]} \vdash$$
$$\text{i\_becomes\_cell\_location}(p)(c) \ : \ \text{OM} \to \text{bool} \stackrel{\text{def}}{=}$$
$$\lambda x : \text{OM}.\ \forall v : \text{int}.\ \text{i\_becomes}(c)\, x\, v = \text{put\_int}(\text{heap}(\text{ml} = p, \text{cl} = 0))\, x\, v$$

$$p\text{: MemLoc}, c\text{: OM} \to \text{MyClassIFace[OM]} \vdash$$
$$\text{k\_cell\_location}(p)(c) \ : \ \text{OM} \to \text{bool} \stackrel{\text{def}}{=}$$
$$\lambda x : \text{OM}.\ \text{k}(c)\, x = \text{get\_int}(\text{heap}(\text{ml} = p, \text{cl} = 1))\, x$$

$$p\text{: MemLoc}, c\text{: OM} \to \text{MyClassIFace[OM]} \vdash$$
$$\text{k\_becomes\_cell\_location}(p)(c) \ : \ \text{OM} \to \text{bool} \stackrel{\text{def}}{=}$$
$$\lambda x : \text{OM}.\ \forall v : \text{int}.\ \text{k\_becomes}(c)\, x\, v = \text{put\_int}(\text{heap}(\text{ml} = p, \text{cl} = 1))\, x\, v$$

$$p\text{: MemLoc}, c\text{: OM} \to \text{MyClassIFace[OM]} \vdash$$
$$\text{MyClassFieldAssert}(p)(c) \ : \ \text{bool} \stackrel{\text{def}}{=}$$
$$\begin{aligned}&\text{ObjectFieldAssert}(p)(\text{super\_Object}(c)) \wedge \\ &\forall x : \text{OM}.\ \ \text{i\_cell\_location}(p)(c)\, x \wedge \\ &\qquad\qquad \text{i\_becomes\_cell\_location}(p)(c)\, x \wedge \\ &\qquad\qquad \text{k\_cell\_location}(p)(c)\, x \wedge \\ &\qquad\qquad \text{k\_becomes\_cell\_location}(p)(c)\, x\end{aligned}$$

The predicate MyClassFieldAssert binds all this together (including the assertion that all the fields in `Object` are appropriately bound to their memory locations). When reasoning about JAVA programs, an assumption is used that MyClassFieldAssert is true, *i.e.* it is assumed that every field is stored at some unique and known cell location. In the semantic description of class `MySubClass` the field `j` gets assigned the cell location 2 in the list ints. MySubClassFieldAssert is defined as: MyClassFieldAssert and `j` is stored at cell location heap(ml $= p$, cl $= 2$) in the list of ints in the memory model. A similar thing is done for `i` in `AnotherSubClass`. This field is stored at cell location get_int(heap(ml $= p$, cl $= 3$)) in the list of ints, and thus completely independent of the "old" `i` field. In a "correctly modeled" instance of `AnotherSubClass` (*i.e.* satisfying AnotherSubClassFieldAssert), stored at memory location p, the variables can be looked up as follows.

variable	access
`i` from `MyClass`	get_int(heap(ml= p, cl= 0))
`k` from `MyClass`	get_int(heap(ml= p, cl= 1))
`j` from `MySubClass`	get_int(heap(ml= p, cl= 2))
`i` from `AnotherSubClass`	get_int(heap(ml= p, cl= 3))

All this bookkeeping is handled by the LOOP compiler.

2.6.8 Method bodies

The next step is to translate the method bodies into a type theoretic description. As an example, the translation of the method body of the method `n` in `MySubClass` is discussed.

Recall the JAVA code for this method.

– JAVA

```
int n (byte b) {
  m(b, 3);
  return i;
}
```

The translation of this method body into type theory is given in Figure 2.15. It takes several parameters:

- c, representing the current object (with appropriate method and field lookup);
- sc, representing the coalgebra that should be used for calls to `super` (with appropriate method and field lookup, it is not used in this example);
- a memory location p, denoting where the contents of the fields of the object are stored, and
- the argument b.

The translated method body starts by allocating cell locations on the stack for the special variables ret_n and par_b – with appropriate assignment operations – representing the return variable and parameter. If a method has local variables, these are formalised in the same way. Before the "real" body is executed, the stack top is increased by one, and the value of the parameter (and, possibly the initial values of the local variables) is assigned to the appropriate variable (*i.e.* to par_b). We choose to have the parameters set on the stack in the method body, instead of before the method call (by the callee), since an assignment operation on the parameters is available in the method body. If the callee would do this assignment, *both* the lookup and the assignment operations of the method would have to be passed on to the method body, and this would make reasoning more complicated. Either one has to reason about the callee, including the allocation of the parameters on the stack, thus loosing abstraction, or one has to reason about a body with the lookup and assignment operation as parameter, which would require extra assumptions about these parameters.

After execution of the whole body, the stack top is decreased again, freeing the memory used for the parameters, local variables and return variable. This cell on the stack at the stacktop corresponds roughly to the activation record or frame [WM95] of a method call.

– TYPE THEORY

p: MemLoc,
b: byte,
c: OM → MySubClassIFace[OM],
sc: OM → MySubClassIFace[OM] ⊢

n_bytebody$(c)(sc)(p)(b)$: OM → ExprResult[OM, int] $\stackrel{\text{def}}{=}$

λx: OM.
(LET ret_n: OM → int = get_int(stack(ml = stacktop x, cl = 0))
ret_n_becomes: OM → int → OM =
put_int(stack(ml = stacktop x, cl = 0))
par_b: OM → byte = get_byte(stack(ml = stacktop x, cl = 0))
par_b_becomes: OM → byte → OM =
put_byte(stack(ml = stacktop x, cl = 0))
IN
(CATCH-EXPR-RETURN(
stacktop_inc;
E2S(A2E(par_b_becomes)(const(b)));
m_byte_int(F2E(par_b))(const(3))(c);
E2S(A2E(ret_n_becomes)(F2E(i(c))));
RETURN)
(ret_n) @@
stacktop_dec) x)

Figure 2.15: The body of method n in `AnotherSubClass` in type theory

Since n is a non-`void` method, it returns a ExprResult in our semantics. As explained (on page 20), for every non-`void` method, the method body is wrapped up in a CATCH-EXPR-RETURN statement. The decrementing of the stack top is the only thing that remains to be done after evaluation of CATCH-EXPR-RETURN. The order in which the CATCH-EXPR-RETURN, stacktop_inc and stacktop_dec are executed may seem a bit strange, but it is necessary to ensure that ret_n is not erased too early. Notice that stacktop_inc cannot be put before CATCH-EXPR-RETURN, because that would require composition of statements and expressions.

Decreasing the stack top also has to be done if the method terminates abruptly, because of an exception. Therefore a special deep composition operation @@ is used, which also has an effect if its first argument returns an abnormal state. This operation is defined as follows.

– TYPE THEORY

e: Self → ExprResult[Self], f: Self → Self ⊢

e @@ f : Self → ExprResult[Self] $\stackrel{\text{def}}{=}$

λx: Self. CASE e x OF {
| hang ↦ hang
| norm y ↦ norm (ns = $f(y$.ns), res = y.res)
| abnorm a ↦ abnorm(es = $f(e$.es), ex = e.ex)

The operation @@ is overloaded, so that it also works with a statement as first argument in case of `void` method bodies.

All the functions discussed so far, occurring in the type-theoretic description of the body of method `n`, do not have an immediate counterpart in the JAVA code. They all explicitly show aspects of the semantics of JAVA that are implicit in the JAVA code and in the execution model of JAVA. The only part of the translation that has not been discussed so far, is the translation of the actual body, *i.e.* the method call to `m`, followed by the `return` statement. Recall how this method body was translated.

```
m(b, 3);
return i;
```

becomes

$$\begin{array}{l}\mathsf{m\_byte\_int}(\mathsf{F2E}(\mathsf{par\_b}))(\mathsf{const}(3))(c)\,;\\ \mathsf{E2S}(\mathsf{A2E}(\mathsf{ret\_n\_becomes})(\mathsf{F2E}(\mathsf{i}(c))))\,;\\ \mathsf{RETURN}\end{array}$$

The method call to `m` is applied to the argument coalgebra c. As explained below (Section 2.6.9), this coalgebra is for appropriate method and field lookup, thus the correct method body is found. Similarly for the field lookup `i`. As explained on page 20, the statement `return` *expr* first evaluates the expression *expr* and assigns this value to a special variable (in this case ret_n), and subsequently RETURN is executed, which brings the program in an abnormal state. Later, CATCH-EXPR-RETURN looks up this return value and returns that as the result of the whole method.

2.6.9 From method call to method body

For each (non-abstract) method, the call (the extraction function) has to be bound to an appropriate method body. Just as for fields, a predicate MethodAssert is defined which connects the call and the body. As an example, the predicate AnotherSubClassMethodAssert is defined below. If a coalgebra satisfies AnotherSubClassMethodAssert this can be interpreted as: there are correct implementations of all the methods in `AnotherSubClass`. Combining this with AnotherSubClassFieldAssert gives a predicate AnotherSubClassAssert, which should be read as: "there is an executable, working implementation of `AnotherSubClass`". When reasoning about an object, it is assumed that the appropriate Assert predicate holds.

To model recursive functions appropriately, with possible non-termination, the binding is done by iterating over a bottom element[13]. However, in this thesis we do not consider recursive methods, which allows us to simplify this binding. For each method, if it is non-recursive, a method call can be rewritten to its method body, applied to an appropriately cast coalgebra. This cast coalgebra handles late binding, since it ensures that fields and methods are appropriately looked up. The appropriate definitions are given in Figure 2.16.

Suppose that method `n` is called on an instance of `AnotherSubClass`. In our semantics this means that, given c: MemLoc $\to$ OM $\to$ AnotherSubClassIFace and p: MemLoc satisfying AnotherSubClassAssert($c\ p$), the term n_byte(a)($c\ p$)$\ x$ is evaluated. Following the definition above, this call is rewritten to n_bytebody, which is applied to a cast coalgebra AnotherSubClass2MySubClass($c\ p$) (and some other arguments). Within the method body n_bytebody, the method m_byte_int is called, applied to AnotherSubClass2MySubClass($c\ p$). This application is simplified as follows.

[13]Basically, this involves Tarski's least fixed point construction over a flat domain: let $D = \{\mathsf{bot}\} \cup X$ for a set X not containing bot, ordered by $x \leq y \Leftrightarrow x = \mathsf{bot} \vee x = y$. The least fixed point of a monotone function $f\colon D \to D$ is then given by bot, if $\forall n.\, f^n(\mathsf{bot}) = \mathsf{bot}$, and d, if $f^n(\mathsf{bot}) = d \neq \mathsf{bot}$ for some n.

p : MemLoc, c : MemLoc → OM → AnotherSubClassIFace[OM] ⊢

AnotherSubClassMethodAssert(p)(c) : bool $\stackrel{\text{def}}{=}$

$\forall x$: OM. $\forall a$: byte. $\forall b$: int.
m_byte_int(a)(b)(c p) x =
m_byte_intbody (c p)(c p)(p)(a)(b) x ∧

⋮

$\forall b$: byte.
n_byte(b)(c p) x =
n_bytebody (AnotherSubClass2MySubClass(c p))
(AnotherSubClass_sup_MySubClass(c p))
(p)(a) x ∧

⋮

$\forall a$: byte. $\forall b$: int.
MyClass_m_byte_int(a)(b)(c p) x =
m_byte_intbody (AnotherSubClass2MyClass(c p))
(AnotherSubClass_sup_MyClass(c p))
(p)(a)(b) x ∧

⋮

p : MemLoc, c : MemLoc → OM → AnotherSubClassIFace[OM] ⊢

AnotherSubClassAssert(p)(c) : bool $\stackrel{\text{def}}{=}$

AnotherSubClassFieldAssert(p)(c p) ∧
AnotherSubClassMethodAssert(p)(c)

Figure 2.16: Definitions of the predicates relating method calls to method bodies for class `AnotherSubClass`

$$
\begin{aligned}
& \mathsf{m\_byte\_int(F2E(par\_b))(const(3))} \\
& \qquad \mathsf{(AnotherSubClass2MySubClass}(c\ p))\ x \\
\equiv\ & \{\text{method extension function for } \texttt{m} \text{ in } \texttt{MySubClass}, \text{ defined on page 57}\} \\
& \mathsf{m\_byte\_int(F2E(par\_b))(const(3))} \\
& \qquad \mathsf{(MySubClass2MyClass(AnotherSubClass2MySubClass}(c\ p)))\ x \\
\equiv\ & \{\text{method extension function for } \texttt{m} \text{ in } \texttt{MyClass}, \text{ expanding F2E and const}\} \\
& \mathsf{m\_byte\_int(par\_b}\ x)(3) \\
& \qquad \mathsf{(MySubClass2MyClass(AnotherSubClass2MySubClass}(c\ p)))\ x \\
\equiv\ & \{\text{method extraction function for } \texttt{m} \text{ in } \texttt{MyClass}\} \\
& \mathsf{((MySubClass2MyClass(AnotherSubClass2MySubClass}(c\ p))\ x)\mathsf{.m\_byte\_int)} \\
& \qquad \mathsf{(par\_b}\ x)(3) \\
\equiv\ & \{\text{definition of MySubClass2MyClass (similar to AnotherSubClass2MySubClass, see page 56), record simplification}\} \\
& \mathsf{(MySubClass\_sup\_MyClass} \\
& \qquad \mathsf{(AnotherSubClass2MySubClass}(c\ p))\ x)\mathsf{.m\_byte\_int(par\_b}\ x)(3) \\
\equiv\ & \{\text{definitions of MySubClass\_sup\_MyClass, super\_MyClass}\} \\
& \mathsf{((AnotherSubClass2MySubClass}(c\ p)\ x)\mathsf{.super\_MyClass).m\_byte\_int} \\
& \qquad \mathsf{(par\_b}\ x)(3) \\
\equiv\ & \{\text{definition of AnotherSubClass2MySubClass (page 56), record simplification}\} \\
& \mathsf{AnotherSubClass2MyClass}(c\ p)\ x\mathsf{.m\_byte\_int(par\_b}\ x)(3) \\
\equiv\ & \{\text{definition of AnotherSubClass2MyClass (page 56), record simplification}\} \\
& \mathsf{m\_byte\_int(par\_b}\ x)(3)(c\ p)\ x
\end{aligned}
$$

Thus, this call is bound to the method `m` from class `AnotherSubClass`, as should be the case. Notice that, when we are actually reasoning about JAVA programs, with the use of a theorem prover (see Chapter 4), all these rewrites are done automatically, invisible for the user.

Similar reasoning shows that i is bound to the field i in `MyClass`.

$$
\begin{aligned}
& \mathsf{i(AnotherSubClass2MySubClass}(c\ p))\ x \\
\equiv\ & \{\text{unfolding all definitions}\} \\
& \mathsf{i(super\_MyClass(super\_MySubClass}(c\ p)))\ x
\end{aligned}
$$

In conclusion, late binding is realised by binding in subclasses the repeated extraction functions of methods from superclasses to the bodies from the superclasses, but with cast coalgebras.

2.6.10 Method calls to component objects

In this section we consider method calls of the form `o.m()`, where `o` is a "receiving" or "component" object[14]. Field access `o.i` is not discussed explicitly, but it is handled in a similar

[14]The receiving object `o` can be `this`.

– JAVA

```
class UseClass {

    MyClass o1 = new AnotherSubClass();
    MySubClass o2 = new AnotherSubClass();
    AnotherSubClass o3 = new AnotherSubClass();
    MyClass o4 = new MyClass();

  void use() {
    o1.m((byte)3, 4);
    o1 = o4;
    o1.m((byte)3, o2.i);
    o3.m((byte)3, o3.i);
  }
}
```

Figure 2.17: JAVA class `UseClass`

way.

A typical example of a class, containing several components is the class `UseClass` in Figure 2.17. It has four components `o1`, `o2`, `o3` and `o4`. The methods and fields of these components are accessed by so-called qualified expressions, for instance `o1.m((byte)3, 4)` calls the method `m` on the object `o1`. It depends on the run-time class of `o1` which method is actually called.

Suppose that `use()` is executed immediately after initialisation of the class `UseClass`. Then the first time that the method `m` is called (on variable `o1`), the run-time type of the receiver object `o1` is `AnotherSubClass`, while the second time its run-time type is `MyClass`. Thus, different implementations of `m` are executed.

The type-theoretic definition, describing the semantics of the method body of `use()`, is given in Figure 2.18. For the translation of the qualified statements and expressions (in this case: field lookups) auxiliary functions CS2S (for Component-Statement-to-Statement) and CF2F (for Component-Field-to-Field) are used. The first argument to CS2S and CF2F is a function *C*_clg, returning a run-time coalgebra for the receiver object. This function, representing the run-time coalgebra, is built incrementally, by adding rules for each subclass of a class. For example, MyClass_clg is characterised by the following rules.

– TYPE THEORY

$$\mathsf{MyClass\_clg}: \mathsf{string} \to \mathsf{MemLoc} \to \mathsf{OM} \to \mathsf{MyClassIFace[OM]}$$

$$\forall p: \mathsf{MemLoc}.\ \mathsf{MyClassAssert}(p)(\mathsf{MyClass\_clg}(\texttt{"MyClass"})(p))$$

$$\forall p: \mathsf{MemLoc}.\ \mathsf{MyClass\_clg}(\texttt{"MySubClass"})(p) = \mathsf{MySubClass2MyClass}(\mathsf{MySubClass\_clg}(\texttt{"MySubClass"})(p))$$

$$\forall p: \mathsf{MemLoc}.\ \mathsf{MyClass\_clg}(\texttt{"AnotherSubClass"})(p) = \mathsf{AnotherSubClass2MyClass}(\mathsf{AnotherSubClass\_clg}(\texttt{"AnotherSubClass"})(p))$$

```
str : string, p : MemLoc
c : OM → UseClassIFace[OM],
sc : OM → UseClassIFace        ⊢

  usebody(c)(sc)(str)(p) : OM → StatResult[OM] ≝
    λx : OM. (CATCH-STAT-RETURN(
                stacktop_inc ;
                CS2S (MyClass_clg)
                     (F2E(o1(c)))
                     (m_byte_int(int2byte(const(3)))(const(4))) ;
                E2S(A2E(o1_becomes(c))(F2E(o4(c)))) ;
                CS2S (MyClass_clg)
                     (F2E(o1(c)))
                     (m_byte_int(int2byte(const(3)))
                          (CF2F(MySubClass_clg)(F2E(o2(c)))(i))) ;
                CS2S (AnotherSubClass_clg)
                     (F2E(o3(c)))
                     (m_byte_int(int2byte(const(3)))
                          (CF2F(AnotherSubClass_clg)(F2E(o3(c)))(i))))
              @@ stacktop_dec) x
```

Figure 2.18: The body of method `use()` in `UseClass` in type theory

If MyClass_clg is applied to a string `"MyClass"`, we get a coalgebra acting on memory location p, satisfying MyClassAssert. If MyClass_clg is applied to a string `"MySubClass"`, this returns a coalgebra satisfying MySubClassAssert (*i.e.* MySubClass_clg(`"MySubClass"`)(p)), cast to a coalgebra for `MyClass`. Similarly, if MyClass_clg is applied to the string `"AnotherSubClass"`, a coalgebra satisfying AnotherSubClassAssert, cast to `MyClass` is returned. For the other classes, we have similar rules.

All these rules are generated as axioms. If the whole class hierarchy would be known in advance, functions describing these coalgebras could be defined. However, we prefer the translated theories to be extendable, *i.e.* newly defined classes can be translated by the LOOP compiler, using the definitions generated earlier for its superclasses.

These coalgebras are so-called 'loose coalgebras', since they are arbitrary coalgebras about which nothing is known, except that they satisfy certain assertions (but it is not known whether they are *e.g.* final).

These loose coalgebras are used as argument to the functions CS2S and CF2F, which handle the qualified method calls. Figure 2.19 shows the definition of the function CS2S (the definition of CF2F is similar).

Function CS2S has three arguments. As explained, the first argument is the function, producing the loose coalgebra. The second argument is an expression which returns a reference to the component class. The third argument is the statement (parametrised with a coalgebra) that should be executed by the component class. First the reference expression is evaluated, possibly

$$
\begin{array}{l}
\mathit{coalg}\colon \mathsf{string} \to \mathsf{MemLoc} \to \mathsf{OM} \to \mathsf{IFace},\\
\mathit{ref\_expr}\colon \mathsf{OM} \to \mathsf{ExprResult[OM, RefType]},\\
\mathit{statement}\colon (\mathsf{OM} \to \mathsf{IFace}) \to \mathsf{OM} \to \mathsf{StatResult[OM]} \vdash\\
\quad \mathsf{CS2S}(\mathit{coalg})(\mathit{ref\_expr})(\mathit{statement}) \;:\; \mathsf{OM} \to \mathsf{StatResult[OM]} \stackrel{\mathrm{def}}{=}\\
\quad\quad \lambda x : \mathsf{OM}.\ \ \mathsf{CASE}\ \mathit{ref\_expr}\ x\ \mathsf{OF}\ \{\\
\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang}\\
\quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto\ \ \mathsf{CASE}\ y.\mathsf{res}\ \mathsf{OF}\\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{null} \mapsto [\![\texttt{new NullPointerException()}]\!]\\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{ref}\, r \mapsto\ \ \mathit{statement}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{coalg}\,(\mathsf{get\_type}\, r\ (y.\mathsf{ns}))\, r)\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (y.\mathsf{ns})\\
\quad\quad\quad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}(\mathsf{excp}(\mathsf{es} = a.\mathsf{es}, \mathsf{ex} = a.\mathsf{ex}))\ \}
\end{array}
$$

Figure 2.19: The definition of CS2S

returning a reference to an object. In that case, the loose coalgebra is applied to the run-time type of that object and its memory location, returning the representation of the run-time class.

There also exist functions CE2E (for Component-Expression-to-Expression) and CA2A (for Component-Assignment-to-Assignment) with similar definitions. These are used for field access and assignment in components. The function CS2S is used for `void` method calls in components and CE2E is used for non-`void` method calls.

As an example, we look at evaluation of the first statement of the body of the method `use()`, if the method call is done immediately after initialisation of `UseClass`, *i.e.* the run-time class of `o1` is `AnotherSubClass`. Suppose that the fields of `o1` are stored at memory location q at the heap.

$$
\begin{array}{ll}
& \mathsf{CS2S(MyClass\_clg)(F2E(o1}(c))) \\
& \quad (\mathsf{m\_byte\_int(int2byte(const}(3)))(\mathsf{const}(4)))\, x \\
\equiv & \{\text{Definition of CS2S, evaluation of } \mathsf{F2E(o1}(c))\, x, \text{ evaluation of get\_type}\} \\
& \mathsf{m\_byte\_int(int2byte(const}(3)))(\mathsf{const}(4)) \\
& \quad (\mathsf{MyClass\_clg}(\texttt{"AnotherSubClass"})(q))\, x \\
\equiv & \{\text{Definition of MyClass\_clg on } \texttt{"AnotherSubClass"}\} \\
& \mathsf{m\_byte\_int(int2byte(const}(3)))(\mathsf{const}(4)) \\
& \quad (\mathsf{AnotherSubClass2MyClass} \\
& \quad\quad (\mathsf{AnotherSubClass\_clg}(\texttt{"AnotherSubClass"})(q))) \\
& \quad\quad x \\
\equiv & \{\text{Similar deriviation as in Section 2.6.8}\} \\
& \mathsf{m\_byte\_int}(3)(4) \\
& \quad (\mathsf{AnotherSubClass\_clg}(\texttt{"AnotherSubClass"})(q))\, x
\end{array}
$$

Thus, this call will result in execution of the method `m` of class `AnotherSubClass`. Similar reasoning shows that, since after the assignment `o1 = o4`, `o1` has run-time class `MyClass`, the second call `o1.m()` will result in execution of the method `m()` in class `MyClass`. This reasoning also applies to the field lookups `o2.i` and `o3.i`.

2.6.11 Object creation

Finally, the semantics of the creation of new objects will be discussed. Explicit creation of objects is done by a class instance creation expression [GJSB00, §§15.8] (or invocation of the `newInstance` method of class `Class`). The class instance creation process consists of the following steps [GJSB00, §§12.5].

- One cell of memory space is allocated for all fields, including those from the superclass.
- All fields are initialised to their default values.
- The appropriate constructor function (depending on the number and types of the arguments) is called.
- If the constructor begins with an explicit constructor invocation, then this constructor is processed (recursively).
- Otherwise, the constructor of the superclass is processed (recursively). This superclass constructor may be given explicitly, or implicitly.
- Next, the fields are initialised in the order in which this is done in the program code (if any).
- The remainder of the body of this constructor is executed.
- A reference to the newly created object is returned.

This process is formalised as follows. First of all, for each class `C` a function new_C is defined, which allocates a new cell on the heap, say at heaptop x, where the contents of the object can be stored, and increments the heaptop. Since there are infinitely many memory cells and the amount of memory in one cell is infinite in our semantics, we do not have to care about `OutOfMemory` exceptions. At the newly allocated memory cell we put a new empty cell, thus making sure that all instance fields are initialised to their default values (see Section 2.5.1). Next the type entry of this new cell is set to the name of the class. The new operation is parametrised with a constructor function. After allocating the new cell, this constructor function is called on the newly allocated object, by using the function this (see Section 2.5.3) and CE2E (see Section 2.6.10).

– TYPE THEORY

str : string, p : MemLoc,
c : OM → MyClassIFace[OM],
sc : OM → MyClassIFace[OM] ⊢

constr_MyClassbody(c)(sc)(str)(p) : OM → ExprResult[OM, RefType] $\stackrel{\text{def}}{=}$

λx : OM.
(LET ret_MyClass : OM → RefType =
get_ref(stack(ml = stacktop x, cl = 0))
ret_MyClass_becomes : OM → RefType → OM =
put_ref(stack(ml = stacktop x, cl = 0))
IN
(CATCH-EXPR-RETURN(
stacktop_inc ;
E2S(A2E(ret_MyClass_becomes(this(p)("MyClass")))) ;
E2S(constr_Object(c)) ;
E2S(A2E(k_becomes(c))(const(3))) ;
E2S(A2E(i_becomes(c))(const(6))))
ret_MyClass)stacktop_dec)x

Figure 2.20: The body of the constructor of `MyClass` in type theory

– TYPE THEORY

$constr$: (OM → MyClassIFace[OM]) → OM → ExprResult[OM, RefType] ⊢

new_MyClass($constr$) : OM → ExprResult[OM, RefType] $\stackrel{\text{def}}{=}$

λx : OM. CE2E(this(heaptop x)($constr$)
(heaptop_inc (put_type
(heaptop x)
(put_empty_heap x (heaptop x))
"MyClass")
(1))

In the translation of the class instance creation expression, we make sure that the appropriate constructor is given as argument. For example, we get the following translation.

[[`MyClass o4 = new MyClass()`]] $\stackrel{\text{def}}{=}$
E2S(A2E(o4_becomes(c))(new_MyClass(constr_MyClass)))

As explained above, before executing the body of the constructor, first another constructor (either from the current or a superclass) has to be called and the fields have to be initialised to their initial value (as explicitly stated in the JAVA code). In our semantics, we choose to do that as the first steps in the constructor body. Figure 2.20 shows the semantics of the body of the constructor of `MyClass`.

```
class MyClass {

  int i;
  int k = 3;
  ..
  MyClass() {
    i = 6;
  }
}
```

Before anything else is done, the reference to the newly created object is assigned to the return value of the constructor ret_constr_MyClass. There is no explicit constructor invocation in this constructor, thus the next step is to invoke the default constructor from its superclass `Object`. Then, the fields are initialised to their initial values. In this case, there is only one field, namely `k` which has an initial value (namely 3). Thus, we get an assignment which sets `k` to 3. Then the 'visual' body of the constructor is evaluated, setting `i` to 6.

2.7 Conclusions and related work

This chapter discusses (a significant part of) a semantics for JAVA. The first sections describe the so-called semantic prelude, the static part of the semantics, which is the same for all JAVA programs. This semantics resembles the semantics of other imperative languages. We aim at describing the whole language, with all its messy details and not just an idealised subset. Interesting aspects of the semantics are its capability to deal with abruptly terminating statements (including exceptions) and the underlying memory model. The last section of this chapter describes the semantics that is used for classes and objects. This semantics is based on coalgebras. Every class gives rise to a collection of definitions and rewrite rules, capturing its semantics. In the LOOP project, this semantics is generated automatically for each class.

There are several references to other semantics for JAVA. A semantics of JAVA in the context of abstract state machines is given by [BS99]. This semantics is described at a very high and abstract level, which allows to leave out many details, in contrast to our semantics which spells out all details. It would require much adaptation to make their semantics suitable for a theorem prover, because theorem provers typically require all these details.

Much work on JAVA aims at (tool-assisted) reasoning about JAVA. Here one should distinguish between work aimed at (1) reasoning about JAVA as a language, and work aimed at (2) reasoning about programs written in JAVA. In the first category there is work on, for example, safety of the type system [ON99, Sym99], or bytecode verification [Pus99, Qia99, HBL99]. The work presented in this thesis falls in the second category. Related work in [PHM99, PHM98] describes the JAVA semantics at a more abstract level, which tries to exploit commonalities in behaviour. In particular, they use a more abstractly described object store, in contrast to our memory model which is very concrete. In its current state, their semantics does not cover abrupt termination (caused by exceptions for instance).

The semantics of inheritance – as a basis for reasoning about classes – is a real challenge, see *e.g.* [Car88, Mit90, CP95, Jac96, HNSS98, NW98]. There is a whole body of research on encodings of classes using recursive or existential types, in a suitably rich polymorphic type

theory (like $F^{\omega}_{<:}$ or $F_{<:}$). Four such (functional) encodings are formulated and compared in a common notational framework in [BCP97]. But they all use quantification or recursion over type variables, which is not available in the higher order logic (comparable to the logics of PVS and ISABELLE/HOL) that is used here. The setting of the encoding in [NW98] is higher order logic with "extensible records". This framework is closest to what we use (but is still stronger). Also, an experimental *functional* object-oriented language, without references and object identity is studied there. This greatly simplifies matters, because the subtle late binding issues involving run-time types of objects (which may change through assignments, see Section 2.6.10) do not occur. Indeed, it is a crucial aspect of *imperative* object-oriented programming languages that the declared type of a variable may be different from – but must be a supertype of – the actual, run-time type of an object to which it refers. Our semantics of inheritance works for an existing object-oriented language, namely JAVA, with all such semantical complications.

Chapter 3

Interactive theorem provers:
PVS and Isabelle

An interactive theorem prover is a computer system that allows the user to enter logical formulae and subsequently prove their correctness. The proving is done as follows: the system keeps track of the open goals, *i.e.* the goals that remain to be proven, and the user gives commands that should be applied to the various goals. Thus all the proving is done by the user, but the systems ensures that all the the rules are applied correctly, without small mistakes slipping through. The system provides an input language in which the formulae can be written and a proof engine, which applies the logical inferences that the user wishes to apply.

Already since ancient times, a language exists, called mathematics, in which logical formulae can be written down and proven. Without the help of interactive theorem provers, hundreds of interesting theorems have been proven, in a nice and elegant way. Thus, it is a good question whether there is actually a need for interactive theorem provers.

The answer to this question is yes, certainly in a computer science setting. Typically, verifications in a computer science setting are very large, with many different, but similar cases. All these cases have to be distinguished and handled carefully, so that subtle differences are not overlooked. Interactive theorem provers are good in doing these large verifications, which involve much bookkeeping and repetition in the various subgoals. If such a verification is done by hand (*i.e.* with pen and paper) it is easy to make small mistakes: forgetting a proof obligation, introducing typing errors *etc.* In these large verifications one is often not really interested in how the proof is constructed. Most steps are straightforward applications of standard proof steps and there are only a few interesting steps. In the end, it is only important that the verification is done, not how it is done.

A typical example of such large verifications is the field of program correctness. Much of the work here is routine work, applying simple (rewrite) rules. A computer system is much better and faster at this than a human. There are usually only a few points in the program verification where user intervention is necessary and choices have to be made, the rest of the proof can be done by the automatic pilot, so to speak. In program verification, speed is also an important factor. It is not possible to wait a year, until a program is completely verified. The use of a theorem prover may significantly increase the "proof throughput", by providing a high degree of automation and applying big proof steps at once.

Interactive theorem proving is not only applied in the field of program verification (in all its variations). It also has been used for more theoretical applications, including (re)verification of

many mathematical theorems. The reason for doing this, is the rigid correctness that potentially can be offered by an interactive theorem prover [Bar96].

Over the years, an overwhelming number of different (interactive) theorem provers have become available (see *e.g.* the *Database of Existing Mechanised Reasoning Systems*, with more than 60 references to theorem provers [DAR]). Many of these focus on first order logic and fully automated proving. Here we restrict our attention to interactive theorem provers for higher order logic. A logic is called higher order if it allows quantification over propositions and predicates.

The existing theorem provers for higher order logic can be classified in several categories, based on the design philosophy and the style of proving. We will briefly discuss these categories, and describe the most well-known theorem provers in these categories. In the rest of this chapter we discuss two theorem provers in more detail: namely PVS [ORR+96] and ISABELLE [Pau94].

- **Type-theoretic theorem provers** There are several theorem provers that are based on type theory. They use the Curry-Howard correspondence of propositions as types, proofs as terms, which means that theorems are seen as types which are true if there is an inhabitant of this type. Thus proof construction is the same as constructing a term of this type. Of course, the specification languages of these systems provide an extensive type system, typically including dependent types. The theorem provers provide so-called tactics to the user, which can be used to build up such terms. These terms (also known as proof objects) can be checked later, by an independent proof checker. When the term inhabits the type, it is a proof of the corresponding theorem. As the proof is checked after construction, the results of the tactics need not be fully trusted. The same approach can be taken in theorem provers in the other categories as well, but the Curry-Howard correspondence provides a natural way to record the proof as a lambda term, which can easily be checked. The indepedent checker can be small (only a few pages) and the verification of this checker can thus easily be established by hand. Well-known examples of theorem provers in this category are AUTOMATH [Bru70], NUPRL [CAB+86], LEGO [LP92] and COQ [BBC+99]. AUTOMATH is one of the first theorem provers. It has been used to prove a large collection of mathematical theorems. NUPRL has been used in the verification of several software systems. LEGO is mainly a theoretical system, which does not provide powerful tactics. Constructing a large proof in LEGO is a monks work, as every detail of the proof has to be spelled out to the system completely. The COQ system provides much more user support and also has been applied to several non-trivial examples, for example verification of JAVACARD programs [BDJ+00], hardware verification [CGJ99] and geometric modelling [PD98].

- **The LCF style provers** One of the first theorem provers was the LCF prover (for Logic of Computable Functions) [GMW79]. The basic idea behind it is that theorems are an inductive datatype, whose terms only can be obtained using its constructors. These constructors correspond to basic logical inferences. All other proof strategies are build in terms of these constructors. This inductive datatype forms the kernel of the system, everything else is build on top of this. As there are only a few basic inferences, only the correctness of the inference steps in the kernel have to be checked. All other proof steps are correct by construction, since they are build on top of correct steps. The LCF prover is programmed in ML, and this language is also available to write the proof strategies.The LCF prover also introduced the term tactics and the idea of backward proving. A user

starts with the desired goal and by applying tactics, it breaks the goal down into smaller subgoals. This is contrary to the way mathematical proofs are traditionally written down. The most well-known examples of theorem provers in the LCF tradition are ISABELLE and HOL [GM93]. The remainder of this chapter discusses ISABELLE in full detail. The HOL system is widely used and has been applied to all kinds of verifications. It provides a high degree of automation and powerful proof tactics. One of the most impressive applications of the HOL systems is the formalisation of real and floating point numbers [Har98]. Other applications of the HOL system are for example in the field of hardware verification [Kro99] and program semantics [Nor98] and verification of distributed programs [Pra95].

- **Declarative proof systems** Declarative theorem provers are quite different from the other theorem provers in the way that proofs are constructed. The other systems all provide backward proving, but in a declarative systems, proofs look more like the traditional proofs. The user gives intermediate results and hints why these intermediate results can be constructed, the system checks that the hints really establish the intermediate result. Typical mathematical proofs can straightforwardly be formalised in such a way. Declarative proof systems are not very widely used. There are some systems under development: for example the DECLARE system [Sym99] and the ISAR system [Wen99]. The last one is a variant of ISABELLE. A much older declarative proof system is MIZAR [Rud92]. Many mathematical theorems have been formalised within this system, but it is only used in a very small community.

- **The pragmatic system** To conclude there is one important theorem prover that does not fall into one of these categories, but nevertheless should be mentioned, namely PVS. PVS is a typical example of a pragmatic system, where efficiency is more important than correctness. The PVS theorem prover provides a collection of powerful primitive inference procedures that are used to construct proofs. These primitive inferences include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic. Their implementations are optimised for large proofs: for example, propositional simplification uses BDDs, and auto-rewrites are cached for efficiency.

When we started working on verification of JAVA programs an important question was which theorem prover to use for the verifications. Introductory papers on particular theorem provers usually emphasise their strong points by impressive examples. But, if one wishes to start using one particular theorem prover, this information is usually not enough. To make the right choice, one should also know (1) which are the weak points of the theorem prover and (2) whether the theorem prover is suited for the application at hand. The choice of a theorem prover is very important: it can easily take half a year before one fully masters a tool and is able to work on significant applications.

We chose PVS and ISABELLE as the basis for our work, because both are known as powerful theorem provers for higher order logic, which have shown their capabilities in non-trivial applications. Both PVS and ISABELLE are complex tools and it takes time to learn to work efficiently with them.

Our experiences with these two theorem provers formed the basis for a comparison [GH98]. This comparison can be seen as an initial impetus to a consumer's report for theorem provers. A useful consumer's report for theorem provers should not summarise the manuals, but be based on practical experience with the tools. The comparison discusses several important aspects from

a user's perspective, both theoretical (*e.g.* the logic used) and practical (*e.g.* the user interface). At the end of the report, there is a list of criteria on which the theorem provers are compared. Such a consumer's report can be interesting for both new and experienced users. They can assist in selecting an appropriate theorem prover, but they also can help to gain more insight in various existing theorem provers, including the proof tool one is usually working with.

This chapter, which is an elaboration and update of [GH98], discusses and compares several aspects of PVS and ISABELLE in detail. As both systems are complex, it is impossible to take all features into account. Our description of the important features of these theorem provers is to some extent subjective. We are aware that theorem provers change in time and that this description only can have temporary validity. However, we hope it has some influence on the direction in which theorem provers are developing.

This chapter is organised as follows. First, Section 3.1 describes what the characteristic aspects of a theorem prover–from a user's perspective. Then Section 3.2 describes PVS and Section 3.3 describes ISABELLE. Based on these descriptions a comparison between the two theorem provers is made in Section 3.4. Finally, we conclude with conclusions and related work.

This chapter is based on experiences with PVS version 2.3 and ISABELLE99.

3.1 Theorem provers from a user's perspective

To describe a theorem prover, it should first be clear which aspects of a theorem prover are important. This section briefly describes these aspects and discusses why they are important. The more detailed description of PVS and ISABELLE is structured along these lines. The division is somewhat artificial, because strong dependencies exist between the various parts, but it is helpful in comparing the two systems. Also, it helps in pinpointing what the essential characteristics of a theorem prover are. The emphasis here is on aspects that are important from a users' perspective.

The first aspect that characterises a theorem prover is the **logic and type theory** that is used by the tool. Within the LOOP project, we restrict ourselves to (extensions of) typed higher order classical logic. The type theories and logics of both PVS and ISABELLE/HOL are a superset of the (simple) type theory and higher order logic that is used to describe the JAVA semantics in Chapter 2. For all the type-theoretic constructs it is explained how they are available in PVS and ISABELLE/HOL.

Strongly related with the logic is the **specification language**. However, it involves more than the logic alone, *e.g.* the exact notations to be used (or how the user can specify his/her own syntax) and the available module structure are also part of the specification language. Nevertheless, the logic and specification language of a theorem prover should always be considered together. The specification language is important for the usefulness of a theorem prover, because a significant part of a verification effort boiles down to specifying what one actually wishes to verify. It is not very useful to have a fully verified statement, if it is not clear what the statement means.

The next aspect that is distinguished is the **prover**. An important issue for the prover is the set of available proof commands (tactics, *i.e.* possible proof steps). Within the LOOP project, much attention is paid to a high degree of proof automation, by automatic rewriting. However, in interactive verification, the possibility to control the proof is also very important. If a statements cannot be proven automatically, the user should be able to guide the theorem prover in the

right direction (and then employ the automatic proving capabilities again). Usually, a user can program his/her own tacticals or proof strategies, which are functions which build new proof commands, using more basic ones. A sophisticated tactical language significantly improves the power of a prover, since it allows the user to encode complicated proof structures. Also related with the proving power of a theorem prover is the availability of **decision procedures** (such as for linear arithmetic and for abstract data types). Decision procedures can do many easy 'calculations' for the user, thus allowing him/her to concentrate on the essential parts of the proof.

Another aspect is the **architecture** of the tool, in particular whether there is a small kernel which encapsulates all basic logical inferences. When the code of the kernel is available (and small) it is possible to convince oneself of the **soundness** of the tool. For a system with a large and complex kernel, this might be more complicated. Typically, in a system with a small kernel, decision procedures are built on top of the kernel, thus ensuring soundness. The architecture of the tool also has an effect on its efficiency.

Theoretically irrelevant, but very important for the actual use of a tool, are the **proof manager and user interface**. The proof manager and user interface determine *e.g.* how the current subgoals are displayed, whether the proof trace is recorded and how proof commands can be undone. They can assist the user significantly in building up a proof, by taking care of many of the bureaucratic aspects of proof construction. Of course, this does not influence the "computing power" of the tool, but a good proof manager and user interface can significantly increase the effectiveness and usability of a theorem prover.

3.2 An introduction to PVS

The PVS Verification System is being developed at SRI International Computer Science Laboratory at Palo Alto (USA). Work on PVS started in 1990 and the first version was made available in 1993. At the moment, PVS version 2.3 is available. Version 3 is expected to have significant improvements and changes. A short overview of the history of the system can be found in [Rus]. Further information about PVS is available in a language manual [OSRSC99a], a system guide [OSRSC99b], and a prover guide [SORSC99]. PVS is written in LISP and it is strongly integrated with (Gnu and X) EMACS. The source code is not freely available, but the system itself is.

PVS has been applied to several serious problems. A well-known example is its application to the specification and design of fault-tolerant flight control systems, including a requirements specification for the Space Shuttle [CD96]. References to more applications of PVS can be found in [Rus].

3.2.1 The logic

PVS implements classical typed higher order logic, extended with predicate subtypes and dependent types [OSRSC99a, ROS98]. All variables and functions that are used have to be typed explicitly. Below it is briefly discussed how the types and terms of our type theory are expressed in the logic of PVS.

Type variables can be used in PVS by declaring functions in a theory, which is parametrised with type parameters. More information about this approach is given below in the next subsection on the specification language of PVS.

Several built-in types are available in PVS, such as booleans, reals and integers; standard operations on these types are hard-coded in the tool. When shifting from our general type theory to the type theory of PVS the type-theoretic types (constructors) as bool, float and int *etc.* are mapped to these built-in types[1].

Type construction mechanisms are available to build complex types *e.g.* lists, function types, product types, records (labeled products) and recursively-defined abstract data types.

For example, lists are defined in the PVS prelude (which contains the theories that are built-in to the PVS system) using a recursive data type.

– PVS

```
list [T: TYPE]: DATATYPE
 BEGIN
  null: null?
  cons (car: T, cdr:list):cons?

 END list
```

The datatype `list` is parametrised with a type variable `T`. The PVS datatype syntax is very compact. Two constructors are defined, `null` – nil in type theory – and `cons`. Further, two so-called recogniser functions – `null?` and `cons?` – are declared, which determine whether a list is empty or non-empty, respectively. These recognisers are not directly available in our type theory, but can be encoded using the CASE construct. Accessor functions `car` (head in type theory) and `cdr` (tail in type theory) are defined on non-empty lists, returning the head and tail of such a list[2]. Special syntax is available to denote elements in a list. For example, `(:1, 2, 3:)` denotes a list with three elements `1`, `2` and `3`. Many of the standard functions on lists are defined in the prelude.

Product types in PVS are denoted using square brackets, surrounding a comma-separated list of types. Elements inhabiting a product type are denoted using round brackets. Thus, for example `(1,2):[int,int]`. The elements of a product type can be accessed by using the projection functions, where `proj_i` returns the i^{th} element of the product. These projection functions are hard coded into PVS. An update function on products exists, which is denoted using the `WITH` construct. It uses numbers to denote which element of the product should be updated. Since all this is hard coded into PVS, no general definition of `WITH` is available in PVS, but the following lemma illustrates the idea.

– PVS

```
product_update : LEMMA
  FORALL(z : [int, int]) :
    proj_1(z WITH [1 := 3]) = 3 AND
      proj_2(z WITH [1 := 3]) = proj_2(z)
```

Function types in PVS also use square brackets, surrounding an arrow between two types. Currying of functions has to be denoted explicitly, using these square brackets. If arguments to a function are only separated by a comma, this denotes a tuple argument. For example, a function f: int $\rightarrow$ bool $\rightarrow$ real is declared in PVS as follows.

[1] In doing so, aspects of range and precision are ignored.

[2] The use of the names `car` and `cdr` is due to the fact that PVS is implemented in LISP.

– PVS

```
f : [int -> [bool -> real]]
```

On the other hand, a function g: int $\times$ bool $\rightarrow$ real is declared in PVS using a comma separated list.

– PVS

```
g : [int, bool -> real]
```

This is equivalent to a declaration which explicitly denotes the tuple.

– PVS

```
g : [[int, bool] -> real]
```

Arguments in PVS are always surrounded by brackets, which can result in specifications with lots of brackets, if currying is heavily used. For lambda abstraction, a keyword LAMBDA is reserved. Also for functions, an update function exists, denoted with the WITH construct again. It uses a syntax similar to the update function on products. For example the following equality holds.

– PVS

```
function_update : LEMMA
  FORALL(f : [int -> int])(x : int) :
    (f WITH [x := 3]) =
      LAMBDA(y : int) : IF x = y THEN 3 ELSE f(y) ENDIF
```

Notice that the PVS language provides a conditional term IF ... THEN ... ELSE ... ENDIF.

Record types, which are the PVS version of the labeled product types in our type theory, are denoted using special brackets [# and #]. Inhabitants of a record type use (# and #). As an example, consider the type definition of ObjectCell (from Section 2.5.1) in PVS[3].

– PVS

```
ObjectCell : TYPE =
  [#
    bytes? : [CellLoc? -> byte],
    shorts? : [CellLoc? -> short],
    ints? : [CellLoc? -> int_java],
    longs? : [CellLoc? -> long],
    chars? : [CellLoc? -> char],
    floats? : [CellLoc? -> float],
    doubles? : [CellLoc? -> double],
    booleans? : [CellLoc? -> boolean],
    refs? : [CellLoc? -> RefType?],
    types? : string,
    dimlen? : [nat, nat]
  #]
```

[3]The question marks ? are added to avoid name clashes, see Section 4.2.1.

Notice that in our type-theoretic definition a labeled product is used for the entry dimlen, which is left out of this definition, as it would only produce unnecessary overhead. The EmptyObjectCell, which initialises an object cell with Java's default values (see Section 2.5.1), is defined as follows in PVS.

– PVS
```
empty_ObjectCell : ObjectCell =
  (#
    bytes? := LAMBDA(n : CellLoc?) : 0,
    shorts? := LAMBDA(n : CellLoc?) : 0,
    ints? := LAMBDA(n : CellLoc?) : 0,
    longs? := LAMBDA(n : CellLoc?) : 0,
    chars? := LAMBDA(n : CellLoc?) : 0,
    floats? := LAMBDA(n : CellLoc?) : 0,
    doubles? := LAMBDA(n : CellLoc?) : 0,
    booleans? := LAMBDA(n : CellLoc?) : FALSE,
    refs? := LAMBDA(n : CellLoc?) : null?,
    type? := "",
    dimlen? := (0, 0)
  #)
```

There are two syntactic constructs in PVS to form selection terms. Given a variable `x:ObjectCell`, both `bytes?(x)` and `x`bytes?` denote the selection of the `bytes?` entry in `x`.

Also on records, an update function is defined, using the same syntax as before. As an example, the following PVS function `bytes_on_one` returns an object cell where all byte fields are set to 1, and everything else is unchanged.

– PVS
```
bytes_on_one : [ObjectCell -> ObjectCell] =
 LAMBDA(cell : ObjectCell) :
   cell WITH [bytes? := LAMBDA(n : CellLoc?) : 1]
```

Labeled coproduct types can be defined in PVS using datatypes. However, PVS datatypes are more general, since they can also be used to define recursive types, as `list` above for example. A typical example of a labeled coproduct type is the type lift, as defined in Section 2.1. In PVS its definition looks as follows.

– PVS
```
Lift?[X : TYPE] : DATATYPE
BEGIN
  bot? : bot??
  up?(down? : X) : up??
END Lift?
```

Notice the – obvious – similarity with the definition of the `list` datatype before. The datatype is parametrised with a type variable `X`. It has constructors `bot?` and `up?`, and recognisers `bot??` and `up??`. Further there is a destructor function `down?` which is only defined for non-bottom elements. Also a CASE construct exists in PVS, denoted with `CASES ... ENDCASES`; for example, the function defined? can be defined using this construct as follows.

– PVS
```
defined?(l : Lift?[X]) : bool =
  CASES l OF
    bot? : FALSE,
    up?(x) : TRUE
  ENDCASES
```

However, using the recogniser functions an equivalent, but much shorter definition can be given.

– PVS
```
defined?(l : Lift?[X]) : bool =
  up??(l)
```

Notice that these recognisers and destructors only provide nice shorthands, but they do not introduce anything essentially new.

The logic of PVS contains all the usual connectives as AND, OR, IMPLIES and NOT. Also, the (typed) quantifiers FORALL and EXISTS are available. As explained above, a conditional term IF ... THEN ... ELSE ... ENDIF exists. Also, there is a let-construct, LET ... IN ... and a choice operator choose, defined on non-empty sets (which are equivalent to non-empty types in PVS). All these language constructs are built-in to the language. Therefore, efficient decision procedures can be designed for them, but the user cannot get any insight in how they actually work.

Predicate subtypes and dependent types

A typical feature of the type system of PVS is the possibility to use predicate subtypes and dependent subtypes. They are not generally available in theorem provers, in particular not in ISABELLE/HOL. Also in our type theory, they are not present. However, as they can be very useful in writing down a succinct and correct specification, they deserve some attention.

In PVS, a predicate subtype is a new type constructed from an existing type, by collecting all the elements in the existing type that satisfy a certain predicate (see also [ROS98]). One of the most basic examples of a predicate subtype is the type of non-zero-numbers. This type is used in the declaration of the division operator in PVS. The code below is a fragment of the PVS prelude.

– PVS
```
% /= is inequality
nonzero_real: NONEMPTY_TYPE = {r: real | r /= 0}

+, -, * : [real, real -> real]
/ : [real, nonzero_real -> real]
```

When the division operator is used in a specification, type checking requires that the denominator is nonzero. As this is not decidable in general, a so-called Type Correctness Condition (TCC) is generated, which forces the user to prove that the denominator indeed differs from zero. A theory is not completely verified unless all of its type correctness conditions have been proven. In practice, most of the TCCs can be proven automatically by PVS.

If P is a predicate with type `[A -> bool]`, for some type A, then `(P)` denotes the subtype of all elements in A satisfying P, *i.e.* `(P) = {a:A | P(a)}`.

The use of predicate subtypes improves the preciseness of a specification. It enables the user to make very precise specifications, *e.g.* instead of writing a comment that a function should only be applied to non-empty lists, one can reflect this in the type. If the function by accident is called on an empty list, this results in an (obviously) unprovable type check condition. In this way, many semantic errors in a specification can be detected by type checking. Carreño and Miner discuss an example where predicate subtyping improved the specification [CM95].

As mentioned, PVS offers another useful typing facility, namely dependent typing. In PVS, dependent types can only be constructed using predicate subtypes, in contrast to other approaches to dependent typing, *e.g.* Martin-Löf's dependent type theory [ML82], where dependent types can be constructed from equality types. Consider for example the following type definition, which could be used to model arrays.

– PVS

```
Ex_Array[T:TYPE]: THEORY
BEGIN
  Ex_Array: TYPE = [# length : nat,
                      val : [below(length) -> T ]
                   #]
END Ex_Array
```

The type `Ex_Array` is a record with two fields: `length`, a natural number denoting the length of the array, and `val`, a function denoting the values at each position in the array. The domain of `val` is the predicate subtype `below(length)` which contains the natural numbers less than `length`. The type of `val` thus depends on the actual length of the array. This is like a Σ-type in Martin-Löf's type theory.

3.2.2 The specification language

The specification language of PVS is rich, containing many features. Some specific points are discussed below.

- PVS has a **parametrised module** system. A specification is usually divided in several theories and each theory can be parametrised with both types and values. A theory can contain several `IMPORTING` declarations, at arbitrary places, so that a value or type that has just been declared or defined can immediately be used as an argument. Several theories can be put together in one file.

  Polymorphism is not available in PVS, but it is approximated by theories with type parameters. To define a polymorphic function, one can put it in a theory which is parametrised with the type variables of the function. However, this approach is not always convenient, because when a theory is imported *all* parameters should have a value. Thus when a function does not use all type parameters of a theory, the unused types should still be instantiated. This can result in an illogical division in theories. For example, in the PVS prelude, the function composition operator is defined in a theory that has 3 type parameters. The theorem that this operator is associative is stated in a separate theory, because it requires 4 different type parameters.

In our type theory, there is no module structure, but type variables are used. To describe this in the language of PVS, theories and datatypes, parametrised with types are used (see for example the definitions of the datatypes `list` and `Lift?` above). Value parameters for theories are not used in our embedding of the JAVA semantics.

- PVS allows non-uniform **overloading**. This means that different declarations (constants or functions) can have the same name as long as they have different types. For instance, it is allowed to have three declarations `f` in one theory: `f:nat`, `f:[nat -> bool]` and `f:[bool -> bool]`. Different functions in different theories can have the same name too, even when they have the same types. The theory names, often together with the correct instantiation, can be used as a prefix to distinguish between them. Names for theorems and axioms can be reused as well, as long as they are in different theories. Again, qualified names can be used to disambiguate.

  This kind of overloading is used several times in the translation of JAVA classes into type theory, remember *e.g.* the overloading of extraction functions and method extension functions (see Section 2.6.5).

- A theory can start with a so-called **assuming clause**, where one states assumptions, usually about the parameters of the theory. These assumptions are used as facts in the rest of the theory. When the theory is imported and instantiated, TCCs are generated, which force the user to prove that the assumptions hold for the actual parameters.

  A typical example where such an assuming clause is useful, is the following. Chapter 5 describes a Hoare logic, tailored towards JAVA. The rules within this logic have been proven sound *w.r.t* our semantics in both PVS and ISABELLE/HOL. In the total correctness rule for loops a well-founded order is used to show termination. Typically, in PVS this order is an argument of the theory, and it is assumed (in the assuming clause) that it is a well-founded order.

– PVS
```
TotalWhileRule [Self : TYPE, A : TYPE+,
                < : PRED[[A, A]]] : THEORY
BEGIN
  ASSUMING
    wf_A : ASSUMPTION well_founded?[A](<)
  ENDASSUMING
  ...
END TotalWhileRule
```

If this theory is imported, instantiated with a particular well-founded order, the user gets a TCC which forces him to show that the order is indeed well-founded. In the soundness proof for the Hoare logic rules in this theory, the well-foundedness of the order can simply be assumed.

An alternative approach to achieve the same effect is to have the following theory header.

– PVS

```
TotalWhileRule [Self : TYPE, A : TYPE+,
                < : (well_founded?[A])]
               : THEORY
BEGIN
  ...
END TotalWhileRule
```

Again, if this theory is imported, instantiated with a particular well-founded order, the user gets an appropriate TCC.

- As discussed above, **recursive data types** can be defined in PVS. An induction principle and several standard functions, such as map and reduce, are automatically generated from a recursive data type definition. Furthermore, PVS also allows general recursive function definitions. All functions in PVS have to be total on their domain (which can be a predicate subtype): therefore termination of the recursive function has to be shown, by giving a so-called measure function which maps all arguments of the function to a type with a well-founded ordering. During type checking, TCCs are generated that force the user to prove that this measure decreases with every recursive call.

- The **syntax** of the specification language of PVS is not very flexible. Many language constructs, such as `IF ...` and `CASES ...` are built-in to the language and the prover. There is a limited set of symbols which can be used as infix operators; most common infix operators, such as + and <= are included in this set. Sometimes PVS uses syntax which is not the most common, *e.g.* `[A,B]` for a Cartesian product of types `A` and `B` and `(:x,y,z:)` for a list of values `x`, `y`, and `z`.

To illustrate several of the points discussed above, an example PVS specification of the quicksort algorithm is considered.

– PVS

```
% parametrised theory
sort[T:TYPE,<=:[T,T->bool]]: THEORY
BEGIN

 ASSUMING  % assuming clause
   total: ASSUMPTION total_order?(<=)
 ENDASSUMING

 l : VAR list[T]
 e : VAR T

% recursive definitions
% with measures
 sorted(l): RECURSIVE bool =
    IF null?(l) OR null?(cdr(l))
```

```
    THEN true
    ELSE car(l) <= car(cdr(l)) AND sorted(cdr(l))
    ENDIF       % <= infix operator
  MEASURE length(l)


  qsort(l): RECURSIVE list[T] =
    IF null?(l) THEN null
    ELSE LET piv = car(l)
         IN append
             (qsort(filter(cdr(l),
                           (LAMBDA e: e <= piv))),
              cons(piv,
                qsort(filter(cdr(l),
                       (LAMBDA e: NOT e <= piv)))))
    ENDIF
  MEASURE length(l)


  qsort_sorted: LEMMA sorted(qsort(l))

END sort
```

The name of the theory (`sort`) is followed by the parameters of the theory, in this case a type `T` and a relation `<=` on `T`. In the `ASSUMING` clause it is stated that the relation `<=` is assumed to be a total order; the predicate `total_order?` is already defined in the prelude. The `VAR` keyword is used to 'declare' the variables `l` and `e` to have the types `list[T]` and `T`, respectively, unless specified otherwise. When these variables are used in a theorem, a universal quantification is implicitly inserted around the statement. The `sorted` predicate expresses when a list is sorted, with respect to the order `<=`. It is defined recursively, and after the `MEASURE` clause a (well-founded) expression is given which decreases for each recursive call. The function `qsort` sorts a list (using the quicksort algorithm). Here the pivot `piv` is simply the first element of the list `car(l)`. The function `filter(l,p)` removes all elements from the list `l` which do not fulfill the predicate `p`. Finally, the lemma `qsort_sorted` expresses that the quicksort algorithm indeed sorts a list[4]. Notice that this lemma implicitly is universally quantified over `l:list[T]`. The lemma can be proven using induction on the length of the list `l`.

3.2.3 The prover

Proof goals are represented in PVS using the sequent calculus. Every subgoal consists of a list of assumptions $A_1, \ldots A_n$ and a list of conclusions $B_1, \ldots, B_m$. One should read this as: the conjunction of the assumptions implies the disjunction of the conclusions: $A_1 \wedge \ldots \wedge A_n \Rightarrow B_1 \vee \ldots \vee B_m$.

[4]Of course, one also needs to show that the result is a permutation of the original list.

The proof commands of PVS can be divided into three different categories[5].

- **Creative proof commands.** These are the proof steps one provides explicitly when writing a proof by hand. Examples of such commands are `induct` (start to prove by induction), `inst` (instantiate a universally quantified assumption, or existentially quantified conclusion), `lemma` (use a theorem, axiom or definition) and `case` (make a case distinction). For most commands, there are variants which increase the degree of automation, *e.g.* the command `inst?` tries to find an appropriate instantiation itself. Often, these proof-commands also can be fine-tuned by exploring the various argument options.

- **Bureaucratic proof commands.** When writing a proof by hand, these steps often are done implicitly. Examples are `flatten` (disjunctive simplification), `expand` (expanding a definition), `replace` (replace a term by an equivalent term) and `hide` (hide assumptions or conclusions which have become irrelevant, in fact: strengthening the goal or weakening the assumptions).

- **Powerful proof commands.** These are the commands that are intended to handle all "trivial" goals. The basic commands in this category are `simplify` and `prop` (simplification and propositional reasoning). A more powerful example is `assert`. This uses the simplification command and the built-in decision procedures and does automatic (conditional) rewriting. The user can extend the set of rewrite rules by adding appropriate lemmas and definitions to them, using the `auto-rewrite` commands. PVS has some powerful decision procedures, dealing, among other things, with linear arithmetic. The most powerful command is `grind`, which unfolds definitions, skolemnises quantifications, lifts if-then-elses and tries to instantiate and simplify the goal.

Numbers can be used in PVS to specify that a command should work only on some of the assumptions/conclusions, *e.g.* `(expand "f" 2)` expands `f` in the second conclusion. When a specification or theorem is slightly changed (*e.g.* a conjunct is added), the line numbers in the goal often change, which is not very robust. Griffioen [Gri00] suggests a more robust solution, using more elaborate expressions.

When reasoning about (translated) JAVA programs, we try to use as much automation as possible. Appropriate rewrite rules for the semantic prelude can be loaded with one proof command (or tactic or proof strategy). Also, for each translated JAVA class, appropriate rewrite rules are generated, which can immediately be loaded in the rewrite set as well. Using these rewrite rules, proofs for methods without loops, recursion or method calls which are due to late binding, can usually be done by automatic rewriting. Rewriting in PVS is lazy, thus arguments are only rewritten if their values are required. Further, lazy rewriting in PVS in particular means that if the right-hand side of the rewrite rule is a conditional or `CASES` expression, the rule is only applied if the top-level condition rewrites to `TRUE` or `FALSE`. This forces us to do some more user interaction in these cases. Remember for example the method `m` from class `MyClass` in Section 2.6.1.

– JAVA

```
void m (byte a, int b) {  \\ i becomes max(a, b)
  if (a > b) {
```

[5]This division is our own, although it resembles the division made by the PVS developers in [COR+95]. The division is not sharp.

```
        i = a;
      }
  else i = b;
}
```

To prove normal termination of this method, it actually does not matter whether `a > b` holds or not, but to do the proof, this case distinction has to be made explicitly by the user.

A solution would be to add these rules as macro rewrites to the set of rewrite rules in PVS, which enforces that they are always rewritten. However, this introduces the risk of non-terminating rewriting, for example when proving that the following method `f` always terminates normally.

– JAVA

```
void f (int i) {
    if (i == 1) {f(2);}
}
```

PVS provides a limited proof strategy language; containing constructs for sequencing, backtracking, branching, let-binding and recursion. For example, there is a strategy called `then`, which takes two proof commands as arguments and applies them sequentially to the goal. When one wishes to use more complicated proof strategies, for example to write a strategy which inspects the goal, this should be done in LISP.

Proving with one proof command

Efficiency is one of the main design issues of PVS, thus it should be able to do simple proofs automatically and quite fast. Here several examples are considered that illustrate the proving power of PVS. This proving power is significantly improved by the built-in decision procedures for arithmetic. These are used in the following theorem, which is proven almost instantly in PVS by `(ASSERT)`.

– PVS

```
calc : LEMMA
  200 * 36 - 4 + 2 * (36 + 3) =
    500 * 24 - (5 * 6 + 15 * 40) - (400 * 10) - 96
```

Also linear (and some non-linear) arithmetic has standard support in PVS and the next theorem is proven with one single `ASSERT` command again.

– PVS

```
arith : LEMMA
  FORALL (x, z : nat) :
    2 * (x + 24) * (x + z) <=
      49 * (x + z) * x + 60 * (2 * x + z)
```

A well-known [COR+95] example that illustrates the power of the simplification procedures of PVS is the proof of the characterisation of the summation function. The theorem below is proven by a single command `(induct-and-simplify "k")`. This command first applies induction on the goal and then simplifies the remaining subgoals as much as possible.

– PVS
```
sum(k:nat): RECURSIVE nat =
  IF k = 0 THEN 0 ELSE k + sum(k-1) ENDIF
MEASURE k

sum_char: LEMMA sum(k) = k*(k+1)/2
```

3.2.4 System architecture and soundness

The developers of PVS designed their prover to be useful for real world problems. Therefore the specification language should be rich and the prover fast with a high degree of automation (see also [Rus99]).

To achieve this, among other things, powerful decision procedures are added to PVS. However, these decision procedures are hard coded into the system (thus can be considered as part of the large and complex kernel) and sometimes cause soundness problems. Furthermore, PVS once was considered to be a prototype for a new SRI prover. Perhaps for these reasons PVS still seems to contain numerous bugs and frequently new bugs show up. An overview of the known bugs – reported by the users – can be seen on the PVS bug list [PVS].

It would be desirable that the bugs in PVS would only influence completeness and not soundness. Unfortunately, this is not always the case, as several bugs from which `true=false` could be proven have demonstrated [PVS, *e.g.* bug numbers 113 and 160, 161, 275, 331, 345, 371]. And although most bugs do not influence soundness, but still they can be very annoying, in particular if they block progress of the proof process.

Because of the soundness bugs in the past, it is reasonable to assume that PVS will continue to contain soundness bugs. The obvious question thus arises, why there are still so many people using PVS?

Even though PVS contains bugs, it still works correctly most of the time and it is able to find many mistakes in specifications. Also, when constructing proofs, PVS prevents the introduction of small mistakes, which are easily made by humans.

Furthermore, experience tells us that the fixed soundness bugs are hardly ever unintentionally explored, we know of only a single case. Usually, users of a theorem prover have some idea in mind what the proof should look like. If the system suddenly starts to behave in an unexpected way, the user normally understands that there must be something wrong, either with his ideas about the proof or with the system.

Much effort has been put into the development of PVS. For this reason SRI does not make the code of PVS freely available. As a consequence, to most users the structure of the tool is unknown and making extensions or bug fixes is impossible, unless users visit SRI to implement additional features.

3.2.5 The proof manager and user interface

The PVS distribution comes with a standard user interface, which is strongly integrated with EMACS. There also exists a batch mode, which is useful to rerun a large development quickly.

All proofs in PVS are done in a special proof mode. The tool manages which subgoals still have to be proven and which steps are already taken in a proof, so it is not the users responsibility

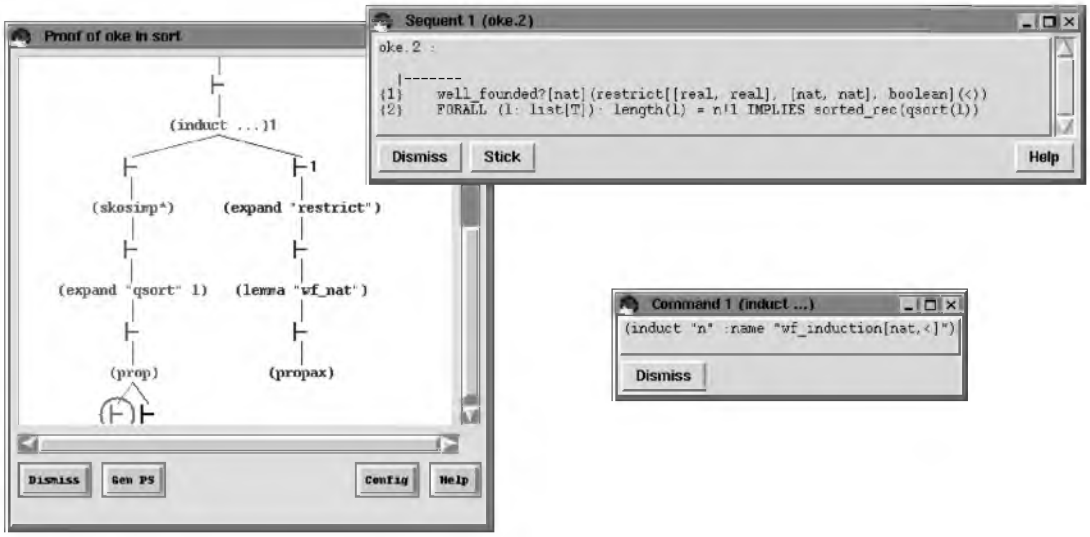

Figure 3.1: Example of a Tcl/Tk proof tree

to maintain the proof trace. Proofs are represented as trees. There is an Tcl/Tk interface which gives a picture of the proof tree (see Figure 3.1). It helps the user to see which branches of the proof are not proven yet. One can click on a turnstile to see a particular subgoal, and also the applied proof commands can be displayed in full detail. Proofs are stored and can be rerun on request, for example to check that a proof is still valid after a change to the theory. It is also possible to step through an already constructed proof, and interactively make changes if necessary. It is possible to tell PVS how many proof steps to take, but it is not possible to tell PVS to run the proof up to a particular point in the proof script (by simply pointing there).

When using a theorem prover, most of the time the theorems and specification are under construction, as the processes of specifying and proving are usually intermingled. The notion of "unproved theorem" allows the user to concentrate on the crucial theorems first and prove the auxiliary theorems later. PVS keeps track of the status of proofs, e.g. whether it uses unproved theorems. Theorems are part of the specifications a user makes in PVS. These specifications are stored in `.pvs` files. The corresponding proofs are kept separately from the specifications in `.prf` files. The user can always ask the system to show the proof of a certain theorem, but standard it is not on the screen.

3.3 An introduction to Isabelle

ISABELLE is being developed in Cambridge, UK, and in Munich (Germany). The first version of the system was made available in 1986. The current version of ISABELLE is called ISABELLE99-1[6]. No major changes are foreseen in new versions. The next version will be able to generate proof objects (in the sense of the type theoretic theorem provers) which can then be checked by an independent checker. As explained above, ISABELLE uses several ideas of the earlier LCF prover [GMW79]: formulae are ML values, theorems are part of an abstract data

[6]As the ISABELLE99-1 version is very recent (from October 2000) this chapter is based on our experiences with ISABELLE99.

type and backward proving is supported by tactics (single proof commands) and tacticals (proof strategies, which are used to build more complex proof commands). The aim of the designers of ISABELLE was to develop a generic proof checker, supporting a variety of logics, with a high level of automation. One of the first texts describing the ideas behind ISABELLE is called the *next 700 provers* [Pau90]. ISABELLE is written in ML, and the source code is freely available.

ISABELLE is used in a broad range of applications: formalising mathematics, logical investigations, program development, specification languages, and verification of programs and systems. References to applications of ISABELLE can be found in [Pfe].

3.3.1 The logic

ISABELLE has a meta-logic, which is a fragment of higher order logic. Formulae in the meta-logic are built using implication $\Rightarrow$, universal quantification $\bigwedge$ and equality $\equiv$. All other logics (the object logics) are represented in this meta-logic. Examples of object logics are first-order logic, the Barendregt cube, Zermelo-Fraenkel set theory and (typed) higher order logic. For higher order logic and ZF set theory, the most elaborate proof support exists.

Here attention is restricted to typed higher order logic (HOL) as object logic. The formalisation of HOL in ISABELLE relies heavily on the meta-logic. HOL uses the polymorphic type system of the meta-logic. In its turn, the type system of the meta-logic is similar to the type system of Haskell. In ISABELLE all function declarations have to be typed explicitly, but for theorems type inference is used (thus the variables occurring in goals do not have to be typed explicitly). A disadvantage of type inference, in combination with implicitly (universally) quantified variables, is that typos introduce new variables, and do not produce an error message. This requires special care from the user. As an example, suppose that one has declared a function `myFunction :: nat => nat`, but that by accident the following goal is typed in: `"myFunction x < myFuntion (x+1)"`. This is internally equivalent to: `"ALL myFuntion. myFunction x < myFuntion (x+1)"`. To detect this error, the user explicitly has to ask for the list of variables (and their types) in the goal.

Implication, quantification and equality are immediately defined in terms of the meta-logic. Together with some appropriate axioms, these form the basis for the higher order logic theory. All other definitions, theorems and axioms are formulated in terms of these basic constructs.

Again, it is discussed how the types from our type theory are represented in ISABELLE/HOL. As the type system of ISABELLE is strongly based on type systems for functional languages, type variables are available. They can be recognised by the fact that a single quote symbol `'` is put in front of their name. As an example a polymorphic constant `arbitrary` is declared as follows.

– ISABELLE

```
arbitrary :: 'a
```

This constant is used later in the definitions of destructor functions on datatypes, in order to handle partiality.

All the type constructs are embedded in the HOL logic, *i.e.* they are build on top of the core logic. Thus, the type constants like nat, int, bool and the recursive type constructor list are all available, with appropriate functions. The fact that all these types are embedded, requires a special syntactic construct for numbers. In ISABELLE/HOL every number literal has to be prefixed by the hash symbol #. Thus, one writes *e.g.* `#3`, to denote the number 3. Natural numbers are

actually defined as Peano numerals. However, the shift between these two representations is handled by ISABELLE.

Functions in ISABELLE/HOL are curried by default. Function application is denoted by juxtaposition. The percentage symbol `%` is used to represent λ-abstraction. The types of the arguments to an Isabelle function are given as a comma-separated list, surrounded by square brackets[7]. If one wishes to give a tuple argument, this tuple type is one of the elements in the list. Thus, f : int $\rightarrow$ bool $\rightarrow$ real is written as follows in ISABELLE.

– ISABELLE

```
f :: [int, bool] => real
```

In contrast, a function g : int $\times$ bool $\rightarrow$ real is declared in Isabelle as follows, where `*` is the Cartesian product constructor.

– ISABELLE

```
g :: "int * bool => real"
```

Notice that the double quote symbol `"` is used in this type declaration. This is necessary, because the `*` symbol is user-defined syntax.

An update function for function types is defined in ISABELLE as follows.

– ISABELLE

```
defs
  fun_upd_def
      "f(a:=b) == % x. if x=a then b else f x"
```

This definition comes with special syntax translation rules, which allow the user to write function updates in this readable format, while they still have a definition build on top of the HOL logic.

As mentioned above, the product type is also defined on top of the HOL logic. Special syntax is given, so that one can write *e.g.* `int * bool` for tuple types, and `(#3, true)` for an inhabitant of this type. Internally, n-product types are represented as $n-1$ nested pairs. Selection functions `fst` and `snd` exist. The third field of a 3-tuple `x` is selected as `snd(snd x)`. However, the third field of a 4-tuple `y` is selected as `fst(snd (snd y))`. Thus, this requires some care from the user.

As in PVS, records are also the ISABELLE version of labeled product types. Records are defined as a special language construct in ISABELLE. As an example, the ISABELLE definition of the object memory type OM is discussed[8].

– ISABELLE

```
record
  OM' =
    heap'top :: MemLoc'
    heap'mem :: MemLoc' => ObjectCell'
    stack'top :: MemLoc'
    stack'mem :: MemLoc' => ObjectCell'
    static'mem :: "MemLoc' => (bool * ObjectCell')"
```

[7]For functions with one argument, these brackets are usually ommitted

[8]Just as question marks are used in the PVS code to avoid name clashes, quote symbols ' are used in the ISABELLE embedding of the JAVA semantics (see also Section 4.2.2). But recall that identifiers starting with a quote ' are used as type variables.

The different entries in the record are listed vertically. An inhabitant of this record type, for example a new object memory, can be defined as follows.

– ISABELLE

```
constdefs
  new_OM :: OM'
  "new_OM == (|
      heap'top = #0,
      heap'mem = % m. empty_ObjectCell',
      stack'top = #0,
      stack'mem = %m. empty_ObjectCell',
      static'mem = %m. (False, empty_ObjectCell')
      |)"

end
```

The order of the entries in the inhabitant should be exactly the same as the order in the record definition, unlike in PVS. An entry of the record type can be selected by applying the appropriate entry name to it, thus *e.g.* `stack'mem x` returns the `stack'mem` entry of `x`, if `x::OM'`. Also a record update function exists, with notation `(| ... := ... |)`. For example, if `x::OM'`, then the same object memory, but with the stacktop reset to 0, is denoted as follows.

– ISABELLE

```
x (| stack'top := #0 |)
```

A feature of records in ISABELLE that is not used here, is their extensability. This forms the basis for an alternative approach to model object-orientation [NW98].

Again similar to PVS, the labeled coproduct types of our type theory are defined using more general recursive data structures. As an example, the definition of RefType in ISABELLE is discussed.

– ISABELLE

```
datatype refType' = Null'
                  | Reference' MemLoc'
```

Thus, the datatype `refType'` is declared with two constructors: `Null'` and `Reference'`. A term tagged with `Reference'` consists of a field of type `MemLoc'`.

The destructor functions can be defined using primitive recursive definitions, as for example this definition of `ref'pos`[9].

– ISABELLE

```
consts ref'pos     :: refType' => MemLoc'

primrec
  "ref'pos (Null') = arbitrary"
  "ref'pos (Reference' pos) = pos"
```

[9]Of course, the function `ref'pos` is not recursive and only uses the pattern match facility of the `primrec` construct. Figure 3.3 shows an example of a real primitive recursive definition.

A construct to make primitive recursive definitions is available for each recursive datatype. More information about recursion is given in the next subsection. Notice that in the case of a null-reference, `arbitrary` is returned. Since nothing is known about this arbitrary element, nothing can be proven about it. In a similar way, recogniser functions can be defined. However, since we avoided using recognisers and destructors in our type-theoretic description of the JAVA semantics, they are also not necessary for the embedding of the JAVA semantics in ISABELLE.

A CASE function is also available in ISABELLE, so alternatively, the function `ref'pos` can be defined as follows.

– ISABELLE

```
constdefs
  ref'pos :: refType' => MemLoc'
  "ref'pos r == case r of
                  Null' => arbitrary
                | Reference' pos  => pos"
```

Finally, constructs such as `if ... then ... else ...`, `let ... in ...` and the choose construct are all available. The choose function is defined axiomatically and forms part of the core of the HOL logic. The other constructs are all defined on top of the HOL logic.

3.3.2 The specification language

The specification language of ISABELLE is inspired by functional programming languages (especially ML). Some specific aspects are discussed.

- The **module system** allows importing multiple other theories, but it does not permit parametrisation. The type parameters of PVS are not necessary in ISABELLE, because declarations can be polymorphic. The value parameters of PVS can be thought of as an implicit argument for all declarations in the theory. Making this argument explicit could be the way to 'mimic' the value parameters in ISABELLE.

- Within different theories, declarations with the same names can be given. These declarations can even have the same arguments. By default, the declaration in the last imported theory is used. If one wishes to use a different declaration, the name should be prefixed with the theory name. Every theory defines a **name space** containing all its declarations, and by explicit mentioning the theory name, the user thus explicitly states in which name space to look for the declaration.

- **Axiomatic type classes** [Wen95, Wen97] are comparable to the assuming clauses in PVS, and type classes in functional programming [WB89]. In a type class polymorphic declarations for functions are given. Additionally, in *axiomatic* type classes, properties that are required for these functions can also be stated. These properties can be used as axioms in the rest of the theory. The user can make different instantiations of these axiomatic type classes, by giving appropriate bodies for the functions and proving that the properties hold. Type classes in functional languages are used to overload functions, for example to overload the + function with different definitions for addition on natural numbers and on integers. The same approach can be used here, but in a limited form, namely only for functions with a single polymorphic type.

– ISABELLE

```
> qsort.rules;
val it =
  ["qsort [] = []",
   "[| ALL x xs. length [y:xs . ~ y <<= x] <
                 length (x # xs);
       ALL x xs. length [y:xs . y <<= x] <
                 length (x # xs) |]
    ==> qsort (?x # ?xs) =
        qsort [y:?xs . y <<= ?x] @
        ?x # qsort [y:?xs . ~ y <<= ?x]"] : thm list
```

Figure 3.2: Conditional rewrite rules generated from the definition of `qsort`

- Another concept which can be used in ISABELLE to assume properties within a theory are **locales** [KWP99]. Locales provide a means to define local scopes, in which abbreviations and assumptions can be made. These abbreviations and assumptions can be used for the proofs within the locale. After closing a locale, the theorems proven in the locale can be used, with the local abbreviations and assumptions added as assumptions to the theorem.

- ISABELLE automatically generates induction principles for each **recursive data type**. The user can give **inductive** and **coinductive** function definitions. There is a special construct to define primitive recursive functions, using the keyword `primrec`. An example of this is the function `ref'pos`, as defined in the previous section. For primitive recursive definitions, termination conditions are automatically proven by the ISABELLE system. For arbitrary recursive definitions, a construct is available to define well-founded recursive functions. The user has to provide an explicit measure from which termination can be proven. From the definition rewrite theorems are generated, which unfold the definition, provided decrease of the measure can be proven for recursive calls. Thus, termination remains to be shown by the user.

  For example, from the definition of `qsort` in Figure 3.3, describing the quick sort algorithm, the theorems in Figure 3.2 are generated. The conditions in the second theorem require the user to show strict decrease of the measure.

- ISABELLE **syntax** can easily be extended. In particular, ISABELLE allows the user to define arbitrary infix and mixfix operators. There is a powerful facility to give priorities and to describe a preferred syntax. For example, for lists a user can write and read *e.g.* `[1,2,3]` while internally this is represented as `(cons 1 (cons 2 (cons 3 nil)))`.

Figure 3.3 shows the quicksort example in ISABELLE syntax. The theory `Qsort` is the union of the theories `HOL`, `List`, `WF_Rel` and the constants and definitions in this file. Remember that type variables start with a quote, in this specification this is `'a`. The constant `<<=` is declared to be an infix operation with priority 65. It is a relation on `'a`. The axiomatic type class `ordclass` is declared as a subclass of the general type class `term`. It has an axiom

– ISABELLE

```
QSort = HOL + List + WF_Rel +      (* theory *)
                                   (* importings *)

consts  (* infix operators *)
  "<<=" :: "['a, 'a] => bool"          (infixl 65)

axclass (* axiomatic type class *)
  ordclass < term
  total_ord "total (op <<=)"

consts  (* primitive recursion *)
  sorted:: "[('a :: ordclass) list] => bool"

primrec
  sorted_nil  "sorted [] = True"
  sorted_cons "sorted (x#xs) = ((case xs of
                                   [] => True
                                 | y#ys => x <<= y) &
                                sorted xs)"

consts  (* well-founded recursion *)
  qsort :: "[('a :: ordclass) list] =>
             ('a :: ordclass) list"

recdef
  qsort "measure size"
    "qsort [] = []"
    "qsort (x # xs) = qsort [y : xs. y <<= x] @
                      (x # qsort [y : xs. ~ y <<= x])"

end
```

Figure 3.3: Specification of the quicksort algorithm in ISABELLE

`total_ord`, which states that that `<<=` is a total order. In this axiom the infix symbol `<<=` is prefixed by `op` to make it behave like a prefix function symbol.

Locales also could have been used to state the assumption that `<<=` is a total order. The definitions would then have been part of the locale, and the final theorems would abstract over these definitions, thus the property holds for all functions satisfying the (recursive) equations, which define `sorted` and `qsort` respectively.

The constant `sorted` is a polymorphic function, where the type parameter `'a` must be in `ordclass`. It is defined as a primitive recursive function, using the special `primrec` declaration. Pattern matching is used to give rules for the definition of `sorted` on the empty list `[]` and on the non empty list `x#xs`. Within the rule `sorted_cons` an extra case distinction on `xs` is made. The constant `qsort` also is a polymorphic function where the type parameter `'a` must be in `ordclass`, but it is defined using well-founded recursion. The `recdef` declaration requires the user to give a measure and rules to define `qsort`. Again pattern matching is used in the definition. The `@` symbol denotes list concatenation. The list comprehension `[y : xs. y <<= x]` should be read as: the list containing all elements `y` of the list `xs`, satisfying `y <<= x`.

3.3.3 The prover

In ISABELLE, every goal consists of a list of assumptions and one conclusion. The goal $[\![A_1; A_2; \ldots; A_n]\!] \Rightarrow B$ should be read as $A_1 \Rightarrow (A_2 \Rightarrow \ldots (A_n \Rightarrow B))$. Notice that $\Rightarrow$ is the implication of the meta-logic.

The basic proof method of ISABELLE is resolution. The operation `RS`, which is used by many tactics, implements resolution with higher order unification. It unifies the conclusion of its first argument with the first assumption of the second argument. As an example, when applying resolution to $([\![?P]\!] \Rightarrow ?P \vee ?Q)$ and $([\![?R; ?S]\!] \Rightarrow ?R \wedge ?S)$, this results in the theorem $[\![?P; ?S]\!] \Rightarrow (?P \vee ?Q) \wedge ?S$.

ISABELLE supports both forward and backward proof strategies, although it emphasises on backward proving by supplying many useful tactics. A tactic transforms theorems into a sequence of theorems. Such a theorem represents the state of a backward proof. If one wishes to prove a goal P, the initial proof state is the (trivial) theorem $[\![P]\!] \Rightarrow P$. The assumptions of this theorem represent the subgoals. Suppose that a tactic transforms the subgoal P into a subgoal Q, then the internal proof state becomes $[\![Q]\!] \Rightarrow P$. The proof is finished when the subgoals have been transformed into **true**, thus the internal proof state is the theorem P.

Many tactics try to find useful instantiations for unknowns in the current goal and the applied theorems. In general there are many possible instantiations, therefore tactics return a lazy list containing (almost) all possible next states of the proof (in a suitable order). When the first instantiation is not satisfactory the next instantiation can be tried with `back()`. This possibility is mainly used by powerful tactics.

The proof commands of ISABELLE can be divided in several categories as well, although these are different from the categories used earlier for PVS.

- **Resolution** forms the basis for a large group of tactics. The standard resolution tactic is `resolve_tac`. It tries to unify the conclusion of a theorem with the conclusion of a subgoal. If this succeeds, it creates new subgoals to prove the assumptions of the theorem (after substitution). **Induction** is done by `induct_tac`, which performs resolution with

an appropriate induction rule. Another variant is `assume_tac`, which tries to unify the conclusion with an assumption.

- **Use of an axiom or theorem** by adding it to the assumption list. There are several variants: with and without instantiation, in combination with resolution etc.

- **Simplifying** tactics for (conditional) rewriting. For every theory a so-called simplification set is built, *e.g.* containing rewrites for the primitive recursive definitions. Simplification tactics try to rewrite goals, using the rewrite rules in this set. The user can add theorems, axioms and definitions (temporarily or permanently).

  ISABELLE's simplifier uses a special strategy to handle permutative rewrite rules, *i.e* rules where the left and right hand side are the same, up to renaming of variables. A standard lexical order on terms is defined and a permutative rewrite rule is applied only if this decreases the term, according to this order. The most common example of a permutative rewrite rule is commutativity ($x \oplus y = y \oplus x$). With normal rewriting (as in PVS) this rule loops, but ordered rewriting avoids this.

  Rewriting in ISABELLE is done eagerly, which means that sub-expressions are always evaluated first, before the top-level expressions. Unfortunately, this increases the risk of non-terminating rewriting. This can be avoided to some extent by using congruence rules. Congruence rules allow a user to force evaluation of a particular subexpression only. Thus, in particular for a conditional expression, simplification of the condition can be enforced first. If this simplifies to either `True` or `False`, only the appropriate part of the condition is rewritten. Using appropriate congruence rules, termination of the method `f` below can be proven in one step, without an explicit case distinction.

– JAVA
```
void f (int i) {
    if (i == 1) {f(2);}
  }
```

However, this does not solve all problems of non-terminating rewriting. Consider for example the ISABELLE theory defined in Figure 3.4. This theory contains two functions `fun1` and `fun2`, with mutually recursive definitions, *i.e.* `fun2` calls `fun1` and `fun1`, which is defined via an axiom, calls `fun2`. Informally, the behaviour of these two functions can be described as follows. The call to `fun1` in function `fun2` is wrapped by a function `apply_once`. This function checks the value of the boolean `x`, if it is false, it is replaced by true and `fun1` is called, otherwise true is returned. If `fun1` is called, it will call `fun2` again, with the argument true. Thus, this time evaluation of `fun2` will terminate. This example may seem constructed, but it actually occurs in the modelling of static initialisation in our JAVA semantics[10].

Suppose that we formally want to prove that evaluation of `fun2 x` always terminates if `check_bool x` does not hold, *i.e.* we have a goal `~check_bool x ==> fun2`

[10]This is not described in this thesis. The basic idea is that static fields of a class are initialised only the first time an instance of this class is made. Therefore, at static initialisation time a boolean is set, which ensures that static initialisation is done only once.

– ISABELLE

```
TrickyRewrite = Main +

constdefs
  put_True :: bool => bool
  "put_True x == True"

  check_bool :: bool => bool
  "check_bool x == x"

  apply_once :: [bool => bool,bool] => bool
  "apply_once f x ==
     (if check_bool x
      then x
      else f (put_True x))"

  wrap :: [bool => bool, bool] => bool
  "wrap f == f"

consts
  fun1 :: bool => bool

constdefs
  fun2 :: bool => bool
  "fun2  == apply_once (wrap fun1)"

defs
  fun1_def  "fun1 == fun2"

end
```

Figure 3.4: Example ISABELLE theory, which results in infinite rewrites

`x = True`. We would like to prove this goal by fully automatic rewriting. Unfortunately, rewriting with all the definitions, including the definition `fun1_def`, makes the ISABELLE simplifier loop. Because rewriting in ISABELLE is eager, the goal is rewritten as follows.

```
    ~check_bool x ==> fun2 x
≡   {definition of check_bool}
    ~x ==> fun2 x
≡   {definition of fun2}
    ~x ==> apply_once (wrap fun1) x
≡   {eager rewriting: rewrite arguments first, definition of fun1}
    ~x ==> apply_once (wrap fun2) x
≡   {definition of fun2}
    ~x ==> apply_once (wrap
                       (apply_once (wrap fun1))) x
≡   {definition of fun1}
    ~x ==> apply_once (wrap
                       (apply_once (wrap fun2))) x
    ⋮
```

Of course, leaving one of the rewrite rules out, in particular leaving out `fun1_def`, avoids that the simplifier loops, but then the goal cannot be proven automatically anymore, because `fun1` has to be rewritten to `fun2` once.

The only way to solve this problem in ISABELLE is to unfold the definition of `apply_once` in `fun2`, and explicitly write the conditional expression in the definition of `fun2`. In this example, the lazy rewriting strategy of PVS clearly has advantages over the eager rewriting strategy of ISABELLE, because a lazy rewriting strategy would evaluate this as follows.

```
    ~check_bool x ==> fun2 x
≡   {definition of fun2}
    ~check_bool x ==> apply_once (wrap fun1) x
≡   {lazy rewriting: definition of apply_once}
    ~check_bool x ==> if check_bool x
                      then x
                      else wrap fun1 (put_True x)
≡   {check_bool x false}
    ~check_bool x ==> wrap fun1 (put_True x)
≡   {definition of wrap}
    ~check_bool x ==> fun1 (put_True x)
≡   {definition of fun1}
    ~check_bool x ==> fun2 (put_True x)
≡   {definition of fun2}
    ~check_bool x ==> apply_once (wrap fun1)
                                 (put_True x)
≡   {definition of apply_once}
```

```
~check_bool x ==> if check_bool (put_True x)
                  then (put_True x)
                  else wrap fun1
                            (put_True(put_True x))
```
≡ {definition of `check_bool`}
```
~check_bool x ==> if put_True x
                  then (put_True x)
                  else wrap fun1
                            (put_True(put_True x))
```
≡ {definition of `put_True`}
```
~check_bool x ==> if True
                  then (put_True x)
                  else wrap fun1
                            (put_True(put_True x))
```
≡ {condition `true`}
```
~check_bool x ==> put_True x
```
≡ {definition of `put_True`}
```
~check_bool x ==> True
```

This evaluation may be less efficient, but it has the advantage that it terminates. This implies that automatic rewriting in PVS is more directly useful for reasoning about (our semantics of) JAVA programs. PVS sometimes requires extra case distinctions, but at least the rewriting does not loop.

- **Classical reasoning** is another powerful proof facility of ISABELLE. There are various tactics for classical reasoning. One of them, `blast_tac`, uses a tableau prover, coded directly in ML. The proof it generates is then reconstructed in ISABELLE. There are also some tactics available which use automatic rewriting in combination with classical reasoning, *e.g.* `auto_tac`, which proves many properties automatically.
- Finally, there are some typical **bureaucratic** tactics, such as `rotate_tac` that changes the order of the assumptions. This can be necessary for rewriting with the assumptions, because this sometimes depends on the order of the assumptions.

Complicated tacticals, *i.e.* functions which combine several tactics can be written in ML, so a complete functional language is available for this purpose. This makes the system very powerful.

Reasoning with meta-variables

A proof goal can contain so-called meta-variables, which can be bound during the construction of the proof. As an example, consider the specification of quicksort (Figure 3.3). Suppose that the axiomatic type class is instantiated with the natural numbers (defining `<<=` as $\leq$ on the natural numbers) and that the definition of quicksort is automatically rewritten. Now the following goal can be stated, where `?x` is a meta-variable.

– ISABELLE

```
Goal "qsort[4, 2, 3] = ?x";
```

When simplifying this goal, the meta-variable is bound to `[2,3,4]` (and the theorem is proven). The theorem is stored as `qsort[4, 2, 3] = [2, 3, 4]`.

This feature makes ISABELLE well-suited for transformational programming [AB96] and writing a Prolog interpreter [Pau94]. Also within the LOOP project, this feature is often employed, not only to "calculate" the result of a method, but also in the application of Hoare logic proof rules.

In PVS, this can be simulated by having an arbitrary variable in the goal. Rewriting then shows what the value of this variable should be. A difference is that in PVS this variable has to be filled in by the user explicitly, and the proof has to be rerun, while ISABELLE binds the meta-variable itself.

Proving with powerful proof commands

Just as for PVS one of the main design goals of ISABELLE is to provide support for efficient reasoning. However, there is an important difference, namely that this is always done on top of the small, correct kernel, thus not compromising on soundness. Therefore, *e.g.* all operations on numbers (naturals and integers) are built on top of this kernel. In PVS arithmetic calculations are done by built-in decision procedures. In ISABELLE/HOL similar properties can be shown, but they are proven using (tractable) simplification. After loading the theories defining the integers, simplification proves the following goal in (almost) zero time. Remember that, for technical reasons, integers are prefixed with a sharp-sign #.

– ISABELLE

```
Goal "(#200::nat) * #36  - #4 + #2 * (#36 + #3) =\
\     #500 * #24 - (#5 * #6 + #15 * #40) - \
\        (#400 * #10) - #96";
```

The simplifier is able to cancel out common summands (and factors). For example, the following goal is proven in one step.

– ISABELLE

```
Goal "#6 + (x :: nat) * x + x * z < \
\        #8 + x  * x + x * z";
```

The variable `x` has to be typed explicitly, to allow ISABELLE to do type inference (since `#6` and `#8` also could denote integers).

An typical example of the power of the classical reasoner of ISABELLE is the following theorem (problem 41 of Pelletier [Pel86]). ISABELLE proves this automatically using the classical reasoner (`Blast_tac`).

– ISABELLE

```
Goal "(ALL z. EX y. ALL x.  \
\        J x y = (J x z & (~ J x x))) --> \
\          ~(EX z. ALL x. J x z)";
```

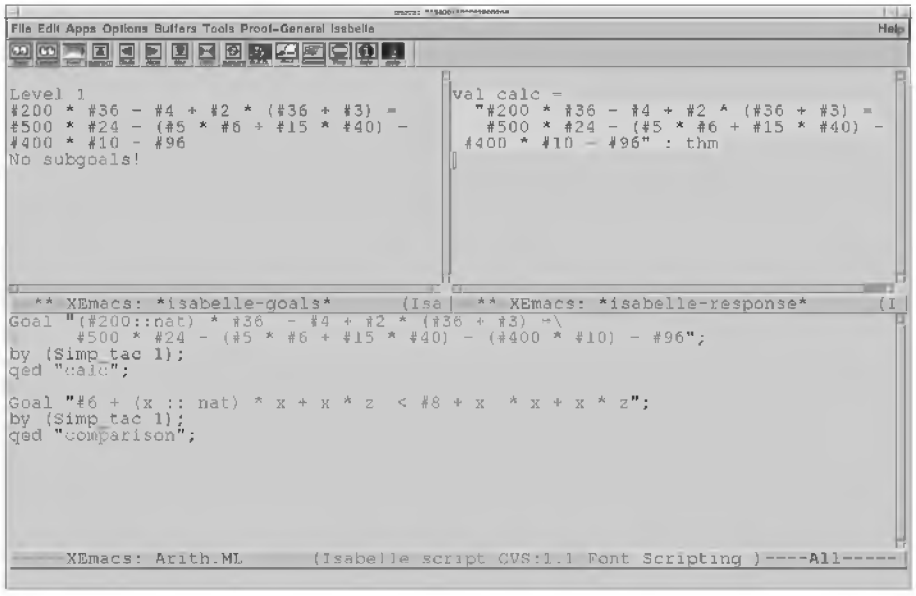

Figure 3.5: A ProofGeneral session

3.3.4 System architecture and soundness

The main objective in the development of ISABELLE was to build a flexible and sound prover, and then to develop powerful tactics and tacticals, built on top of the kernel, so that large proof steps can taken at once. As a result, all powerfull tactics (but excluding the simplifier) make use of the basic inference steps that are part of the kernel. All logical inferences on terms of type `thm` (the theorems) are performed by a limited set of functions. In ML a type can be 'closed', which means that a programmer can express that no other functions than a number of 'trusted' functions are allowed to manipulate values of this type (in this case: theorems). In this way the full power of ML can be used to program proof strategies, and soundness is guaranteed through the interface.

ISABELLE is an open system, which means that everybody can easily add extensions. As long as such extensions do not change the kernel (which should not be possible), soundness is guaranteed by construction.

3.3.5 The proof manager and user interface

The standard "interface" for ISABELLE is a normal terminal window, the so-called xterm interface. In the xterm interface, there is no elaborate proof support. The user has to keep track of everything him/herself (including the undos). The proofs are structured linearly: there is just a list of all subgoals. This stimulates the use of tacticals such as `ALLGOALS`, but it is not so easy to see how "deep" or in which branch one is in a proof. In ISABELLE it is possible to undo an undo (or actually: a choplev, which steps back an arbitrary number of levels, or to a particular level). It is also possible to look at the subgoals at an earlier level, without undoing the proof.

A specification in ISABELLE consists of two kind of files: `.thy` files, which typically contain definitions and axioms, and `.ML` files which contain theorems and their proofs. It is required that the theory name and the file names are the same. In this way, when reloading a

specification, ISABELLE finds the imported theories itself (possibly after setting some search paths). When reloading a specification, also the `.ML` files are reloaded, and all the proofs are rerun again. Thus, reloading files can take quite a while for a non-trivial problem . The user has the possibility to store an image and start working with this image later, thus avoiding rerunning all the proofs. However, a small change in the specification still requires rerunning all the proofs, to restore the image, even if the change only affects a small number of the proofs.

A more elaborate proof manager and user interface are available in the form of Proof General [Asp00], which is a generic user interface for theorem provers. An instantiation of Proof General for ISABELLE exists. ProofGeneral is build on top of Emacs. When working with ProofGeneral, the user gets several buffers: the script buffer (the `.ML` file), the goals buffers, containing all the current subgoals, and the response buffers, showing all the messages from the system (see Figure 3.5). A user can transfer proof commands from the script buffer immediately to ISABELLE. The part of the script that the system already went through, is write-protected to prevent unwanted changes there. The user first explicitly has to undo proof steps before this text can be changed. There is support to step through a proof or jump to a certain point in a script, and colours are available to see which theories and proofs are already loaded. The goals are also displayed using different colours for the variables. If a function name is misspelled, and has become a variable by accident, this is easily recognised by the colouring. Proof General is becoming the *de facto* standard user interface of ISABELLE.

3.4 Comparison I: an ideal theorem prover

In the discussion above, already several weak and strong points of PVS and ISABELLE have been mentioned. This section wraps this up, and gives some ideas what the ideal mixture of PVS and ISABELLE would look like. Later – in Section 8.2 – we will come back to this comparison and discuss which theorem prover is most suited for the LOOP project.

3.4.1 The logic

Our type theory can easily be embedded in both PVS and ISABELLE. The constructs that are used in our type theory are sort of a minimum that a theorem prover for higher order logic should support.

Predicate subtyping and dependent typing give so much extra expressiveness and protection against semantical errors, that it should be supported. The loss of decidability of type checking is easily (and elegantly) overcome by the generation of TCCs and the availability of a proof checker. Overall, the generation of TCCs provides a nice separation of concerns.

The meta-logic of ISABELLE gives the flexibility to use different logics, even in a single proof. However, in our applications, we did not feel the need to use a logic other than HOL and the interference with the meta-logic sometimes complicated matters. If one is only interested in working with higher order logic, then it is not necessary to have other logics around.

The fact that ISABELLE can do type inference is nice, although it might be problematic in combination with predicate subtyping and dependent typing.

In ISABELLE, most language constructs are embedded in the logic. This is a nice approach, since it preserves soundness. On the other hand, if the embeddings are shallow, they are actually only abbreviations and internally enormous terms can be created, which significantly affects the speed of the tool. There are "tricks" to reduce the effect on the run-time speed of the tool,

e.g. wrapping up terms in a datatype. Preferably, the tool applies these tricks standardly, without the user being aware of it.

3.4.2 The specification language

The specification language should be readable, expressive and easily extendible. For function application, we have a preference for the bracketless syntax of ISABELLE. In general, the "functional" style of ISABELLE is nicer to read, especially when currying is used. The flexible syntax of ISABELLE is very nice. The possibility to define translations from and to internal structures, significantly improves the possibility to make readable specifications.

Assuming clauses as in PVS provide a nice and intuitive way to state local assumptions. If a user wants to use theorems that are proven correct with respect to these assumptions, he/she only has to prove once that a particular instantiation satisfies the assumptions. This is in contrast with the locale approach, where the local assumptions become assumptions in all the theorems proven in the locale, and thus have to be discharged every time.

Both PVS and ISABELLE allow the user to define general recursive functions, as long as termination can be proven via a strictly decreasing measure. In PVS special proof obligations (in the form of type check conditions) are generated which force the user to show that the measure function decreases. This gives a nice seperation of concerns: the definition simply can be used and termination is shown independently. In ISABELLE conditional rewrite rules are generated and these two steps become more intermingled. The fact that termination of primitive recursive function is proven immediately in ISABELLE is very nice, since this is the kind of recursion that occurs most.

Further, we prefer to have to the possibility to have several theories in a single file, as is possible in PVS. Dividing a specification in several theories gives more structure. However, for manageability it is preferable not to have to many files. In ISABELLE, where it is not possible to put several theories in one file, this often results in large theories.

3.4.3 The prover

The provers from PVS and ISABELLE are both quite good. A combination of their powers would result in the ideal prover. This ideal prover has powerful proof commands for classical reasoning and rewriting. A tactic returns a lazy list of possible next states, so that (almost) all possible instantiations can be tried. Also, decision procedures (for example for linear arithmetic) are available. Preferably, these decision procedures are not built-in to the kernel, but written in the tactical language, so that they preserve soundness.

The style of the interactive proof commands of PVS is preferred over that of ISABELLE, because it is more intuitive. A structured tactical language, like ML allows the user to write complex proof strategies. The structure of the goal should be well-documented, so that proof strategies are able to inspect the goal.

As discussed above, rewriting is very important in the LOOP project . Both lazy and eager rewriting strategies have their advantages and disadvantages. Preferably, the user should have the possibility to switch between the various rewriting strategies, otherwise it should at least be clear to the user which strategy is used. Congruence rules and ordered rewriting can be used to have more control in the rewriting. Furthermore, it is desirable that the tool gives warnings if it suspects that the rewriting process got stuck in a loop (or reports regularly on progress), so

that the user does not wait forever for an answer, uncertain of whether something useful is still going on.

3.4.4 System architecture

Of course, a theorem prover should be sound. Also other bugs, which might block progress, should not appear. However, also efficiency is an important consideration in the design. If a tool is sound, but too slow, it is not useful for verifications of larger systems. Also, as explained above, even though PVS contains soundness bugs, it is still a great help in specification and verification, since most of the time it works 'correctly'. But of course, ultimately we would like to have a theorem prover without bugs, and especially without soundness bugs. To achieve this goal of a sound theorem prover, a system with a small closed kernel is desirable. The tool should be an open system, of which the code is freely available, so that users can easily extend the tool, on top of the kernel, for their own purposes and (if necessary) implement bug fixes.

The speed of PVS and ISABELLE has not been compared, because the game is not to "run" a proof, but to construct it. This construction consists of building a specification of a problem and proving appropriate theorems. This is hard and depends heavily on the user, his/her experience with the theorem prover *etc.* However, it can be mentioned that the "experienced speed", *i.e.* the waiting time for type checking or executing a (powerful) tactic, of the two tools is comparable. Both for PVS and ISABELLE, the execution of a single command – on a Pentium II 300Mhz – often takes less then a second and hardly ever more than ten seconds.

3.4.5 The proof manager and user interface

The tool should keep track of the proof trace, the user should not be concerned with copying and pasting proof commands. The separate proof files of PVS (the so-called `.prf` files) give a nice seperation of concerns. A user only sees a proof if he wants to, otherwise he is not bothered with it. When reloading older specifications, rerunning of proofs should not be done automatically, only on request.

Proofs are best represented as trees, because this is more natural, compared to a linear structure. The tree representation also allows easy and intuitive navigation through the proof, supported by a visual representation of the tree. When replaying the proof, after changing the specification, the tool can detect exactly for which branches the proof fails, thanks to the tree representation.

As to user interfaces, both ProofGeneral and the PVS user interface are nice and make working with the systems easier, but they still can be improved.

3.5 Conclusions and related work

This chapter describes some important aspects of PVS and ISABELLE which are not in the 'advertising of the tool', but are important in getting a feeling for what the tools are like and what they are able to do. The description consists of the following aspects for each tool: the logic, the specification language, the prover and the proof manager and user interface. These four parts describe the essential components for a theorem prover. Finally, since both PVS and ISABELLE have their weak and strong points, a comparison is made between the tools, resulting in some ideas about what the "ideal" theorem prover should look like.

	PVS 2.3	ISABELLE99/HOL
logic	typed HOL	typed HOL
predicate subtypes	++	not available
dependent predicate subtypes	++	not available
standard syntax	++/+	+
flexible syntax	-	++
module system	++/+	+
polymorphism	-	++
overloading	++	+
abstract data types	++/+	++/+
recursive functions	++/+	++/+
proof command language	+	+/-
tactical language	+/-	++
automation	+	+
arithmetic decision procedures	++	+
libraries	+	++/+
proof manager	++	+ (Proof General)
interface	++	+ (Proof General)
soundness	-	++
upwards compatible	+/-	+
easy to start using	+	-
manuals	+/-	+/-
support	+	++
time it takes to fix a bug	-	?
ease of installation	++	++

Figure 3.6: A consumer report of PVS and ISABELLE

To conclude, Figure 3.6 gives a more detailed list of criteria for judging a theorem prover, filled in for PVS and ISABELLE. This list is not complete and based on the available features of PVS and ISABELLE and our work done with these theorem provers.

We are not the first to compare different theorem provers, but to the best of our knowledge, we are the first to compare PVS and ISABELLE/HOL. Our comparison is not based on a particular example, but systematically treats several aspects of both tools.

A comparison of ACL2, a first-order logic prover based on LISP, and PVS – based on the verification of the Oral Message algorithm – is described in [You97]. HOL is compared to PVS in the context of a floating-point standard [CM95]. In the first comparison, the specification language of PVS is described as too complex and sometimes confusing, while the second comparison is more enthusiastic about it. Gordon describes PVS from a HOL perspective [Gor95]. Other comparisons have been made between HOL and ISABELLE/ZF (in the field of set theory) [AG95], HOL and Coq [Zam97] and Nuprl and NQTHM [BK91]. Three theorem prover interfaces (including PVS) are compared from a human-computer interaction perspective in [MH96].

Chapter 4

The LOOP tool
and its translation of Java classes into PVS and Isabelle

To generate the type theoretic semantic of a JAVA class, as described in Chapter 2, a compiler is used, the so-called LOOP tool. This compiler generates a series of PVS or ISABELLE theories from a JAVA class, describing its meaning, based on the type-theoretic semantics for classes as described in Section 2.6. The LOOP compiler only works for JAVA code that is correct according to the language definition.

The generated theories can be loaded into PVS or ISABELLE, together with the so-called semantic prelude, *i.e.* the general semantics as described in Sections 2.2 – 2.5, which does not depend on the class that is being translated. Subsequently, a user can (try to) prove the desired properties about the original JAVA classes within the interactive theorem prover. Typical examples of properties that a user may want to prove are (non)termination of methods, assertions involving pre- and post-conditions and class invariants. At the moment, the user still has to type in the required properties himself, in the language of the theorem prover, but an extension to the LOOP tool is under development which will make it possible to write the required properties in the JAVA file and to have them translated to PVS or ISABELLE by the compiler.

This chapter is organised as follows. The first section describes the overall architecture of the compiler. Section 4.2 describes the output of the LOOP compiler with respect to the theorem provers PVS and ISABELLE. Section 4.3 describes how one actually proceeds to prove properties about a JAVA program. Then, Section 4.4 describes the automatic verification of some easy (but not straightforward) JAVA programs. Finally, this chapter ends with conclusions and related work.

4.1 Overall architecture of the tool

The LOOP tool is implemented in OCAML [RV98] and has a basic EMACS interface. A graphic description of the overall architecture of the tool can be found in Figure 4.1. Figure 4.2 (page 113) graphically describes the use of the LOOP tool. The LOOP tool starts with a standard lexer and parser, obtained via OCAML versions of `lex` and `yacc`. This parser can take either JAVA, CCSL or JML classes as input. The compiler decides on the basis of the extension of the input file which input type it is. This thesis focuses on JAVA as input language for the tool.

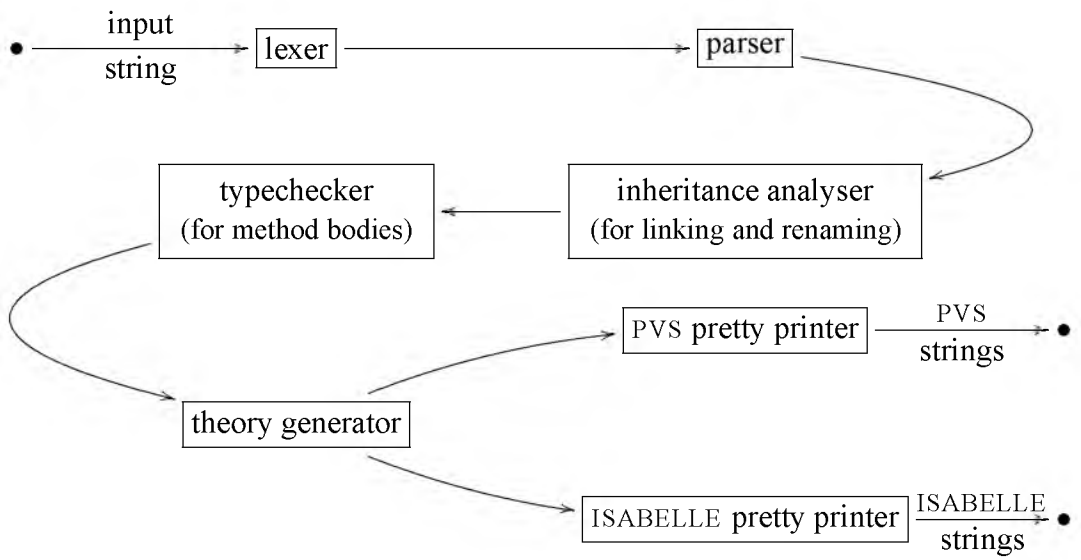

Figure 4.1: The LOOP tool architecture, for JAVA input and PVS/ISABELLE output

The historically first input language for the tool was CCSL (short for Coalgebraic Class Specification Language), which is a class specification language. The first version of the compiler generated PVS theories for CCSL classes. A CCSL class specification consists of declarations of methods, fields and constructors, plus assertions describing their behaviour. More information on this branch of the project can be found in [HHJT98, Tew00].

The language JML (short for JAVA modeling language) [LBR98] is an annotation language for JAVA. An extension of the tool that is currently under development generates appropriate proof obligations based on these JML annotations [BPJ00]. Chapter 6 gives an impression of how such annotations are used and to which proof obligations they give rise. The extension of the LOOP compiler for JML classes, is build on top of the LOOP compiler for JAVA classes.

Via appropriate semantic actions the parser transforms the JAVA classes in the input into some abstract internal representation, using OCAML's data types. This parse tree is modified into an abstract representation of the theories in several compiler passes. First, the inheritance analyser puts appropriate links between classes, and detects name clashes indicating overriding and hiding. Then the method bodies of JAVA method declarations are typechecked, following the standard JAVA typechecking mechanism. This is needed, because at various stages of the translation into PVS/ISABELLE code, the type of a JAVA code fragment that is being translated must be known. Once this is done, logical theories are generated, using some abstract logical representation. Finally, this representation is turned into PVS or ISABELLE code by an appropriate pretty-printer. Whether PVS or ISABELLE theories are generated is decided by a compiler switch.

The PVS and ISABELLE theories that are produced by translating a particular JAVA class consist of the following items.

- Definitions of interface types, translated method bodies, *etc.*, which capture the semantics of the class, based on the semantics as described in Section 2.6.

- Lemmas stating results about these definitions. Many of these lemmas are specifically

generated for automatic rewriting purposes, and contribute to the level of automation that is achieved by the proof tool[1].

- Proofs of these lemmas.

4.2 Reasoning about Java

As mentioned above, the LOOP project aims at reasoning about JAVA classes with the use of a (powerful) theorem prover. As explained in Chapter 3, the assistance of a theorem prover is crucial for the feasibility of the verification. The theorem prover keeps the overview of the verification, and prevents the user from forgetting subgoals. It also can do many simple steps at once, so that the user can concentrate on the crucial parts in the verification.

To shift from the type theoretic semantics of JAVA towards a semantics in the logic of a theorem prover, two steps are needed. First of all, the semantic prelude, describing the basic semantics of JAVA, has to be rewritten in the specification language of the theorem prover[2]. The second step is to adapt the pretty printer of the LOOP compiler, so that it generates a class description in the appropriate output language. Since the type theoretic language that is used in Chapter 2 is (roughly) an intersection of the specification languages of PVS and ISABELLE/HOL, the adaptation is straightforward. However, there are some peculiarities in both specification languages, which require a special treatment.

4.2.1 From type theory to PVS

Suppose that a field or method occurs in a JAVA class, which has the same name as a function in our embedding of JAVA, Within a theorem prover, this name clash would produce a type check error. *E.g.* a variable name for which this could occur is `res`, which would clash with the label res in ExprResult. To avoid these name clashes, in the semantic prelude for PVS, one or more question marks are added to the names of all constants. Since question marks are not allowed in JAVA identifiers, this solves the problem. As an example, the type StatResult is described in PVS as follows.

– PVS
```
StatResult?[Self : TYPE] : DATATYPE
BEGIN
  hang? : hang??
  norm?(ns? : Self) : norm??
  abnorm?(dev? : StatAbn?[Self]) : abnorm??
END StatResult?
```

Another peculiarity of PVS is the need for explicit instantiations. Suppose a function is defined in a parametrised theory (which is used to mimic polymorphism, see Section 3.2.2). If this function is used outside its defining theory, PVS (usually) requires explicit instantiations – sometimes it even needs the full theory name – to allow type checking. As an example, consider the following theory (defining const in the specification language of PVS).

[1] Actually, in the ISABELLE translation the lemmas are generated as axioms at the moment, to avoid the need to generate proofs.

[2] Actually, the project started with describing the JAVA semantic prelude in PVS. Later this semantics prelude has been rewritten in type theory and in ISABELLE.

– PVS
```
ConstantExpression[Self, Out : TYPE] : THEORY
BEGIN

 IMPORTING ExpressionResult[Self, Out]

 const? : [Out ->[Self -> ExprResult?[Self,Out]]] =
   LAMBDA(a : Out) :
     LAMBDA(x : Self) : norm?[Self, Out](x, a)

END ConstantExpression
```

Notice that every time `ExpressionResult`, `ExprResult?` or `norm?` is mentioned in this definition, explicit type instantiations are necessary. Also, when the function `const?` is used, an explicit type instantiation is always needed; for example [[`1.5f`]] is denoted in the PVS translation as `const?[OM?, float](15 * exp (10, -1))`. In order to be able to generate these appropriate type instantiations, the compiler has to keep track of the types of expressions.

4.2.2 From type theory to Isabelle

To avoid name clashes in ISABELLE, the quote-symbol ' is added to the names in the semantical prelude for ISABELLE. This symbol is also not allowed in JAVA identifiers[3]. Name clashes can give unexpected typing problems in ISABELLE, due to the name space mechanism, as described in Section 3.3.2. In the context of inheritance these name clashes cannot be avoided and cause type check problems. As a solution, in many cases the full name of the function (including the theory name) is generated. Consider for example the following JAVA classes.

– JAVA
```
class A {

  int a;
}

class B extends A {

  int b;

  void m() {
    a = 3;
    b = 4;
  }
}
```

The method `m` gives rise to the following definition in Isabelle.

[3]Of course, it would have been desirable to have a common 'distinction'-symbol in PVS and ISABELLE, but question marks are not allowed in ISABELLE function definitions, while quotes are illegal in PVS.

– ISABELLE

```
constdefs
  m'body :: "[(OM' => ((OM')) B'IFace) ,
              (OM' => ((OM')) B'IFace) ,
              MemLoc'] => (OM' => OM' StatResult')"
  "m'body c''  sc''  p'' ==
   (% ((x'' :: OM')) .
     ((catch'stat'return
       ((stacktop'inc ;;
         (E2S' (A2E' (AInterface.a'becomes (B'2'A (c'')))
                                           (const' (#3))) ;;
          E2S' (A2E' (BInterface.b'becomes (c''))
                                           (const' (#4))))))
       @@ stacktop'dec))
      (x''))
   "
```

Thus, reading through all the details that have to be made explicit, the constant `a'becomes` refers to its definition in the interface theory of class `A`, while the constant `b'becomes` originates in the interface theory of class `B`[4].

As already mentioned in Section 3.3.2, the fact that language constructs such as records only are shallowly embedded in ISABELLE, sometimes causes efficiency problems. For example, in the first version of the semantic prelude in ISABELLE, there was a problem with the record type `OM'`, which produced enormous terms. As a solution[5], a single constructor datatype is wrapped around the record definition. A datatype really produces a new type, while a record only creates a type abbreviation. Thus, the theory describing the semantics of the object memory actually starts as follows in ISABELLE.

– ISABELLE

```
record
  primitive_OM' =
    heap'top_in_record :: MemLoc'
    heap'mem_in_record :: MemLoc' => ObjectCell'
    stack'top_in_record :: MemLoc'
    stack'mem_in_record :: MemLoc' => ObjectCell'
    static'mem_in_record :: "MemLoc' =>
                                (bool * ObjectCell')"

datatype
  OM' = OM' primitive_OM'
```

[4]This solution could also have been used to avoid name clashes between function definitions and JAVA fields and methods. However, this would require that the full name is always generated, thus for example `JavaStatement.catch_stat_return` instead of `catch'stat'return`. This would make the translated method bodies even more unreadable than they already are, and would not give any useful extra information. On the other hand, in the context of inheritance, the theory name also gives extra information to the reader.

[5]suggested by Markus Wenzel.

```
consts
  get'OM' :: OM' => primitive_OM'

primrec
  "get'OM' (OM' x) = x"


constdefs
  heap'top :: OM' => MemLoc'
  "heap'top x == heap'top_in_record (get'OM' x)"

...
```

The record type is named `primitive_OM'`. All the entries are provisionally named, by adding `_in_record` to their labels. A datatype `OM'` with only one constructor (`OM'`) is wrapped around this record type. A function `get'OM'` is defined, which forgets the constructor. Functions with the intended label names (*e.g.* `heap'top`) are defined, working on `OM'`. These functions return the appropriate entry of the record. Further, all the definitions can remain unchanged. During proving, the user needs not be aware of this extra layer.

As described in Section 3.3.5, theorems in ISABELLE are stored in `.ML` files, together with their proofs. When loading the theories, all the proofs are rerun. Thus, for all theorems that are generated for rewriting, a proof should be given as well. However, when we started generating output for ISABELLE, the main goal was to get things working first. Therefore, at the moment the rewrite rules are generated as axioms (with an annotation that they actually are theorems). Generating the proofs in ISABELLE is still future work.

4.3 Using the LOOP tool

This section will describe a typical example session of how the LOOP tool is used to reason about a JAVA class. Before starting, one should have available a compiled version of the tool – which is called with the command `run` – and PVS and/or ISABELLE[6].

Figure 4.2 shows the general idea of how to proceed. The LOOP tool is run on some input file (in the rest of this section, it is assumed that this is a JAVA file), and generates a series of logical theories, in the specification language of either PVS or ISABELLE. These logical theories are fed to the appropriate theorem prover, together with the semantic prelude, describing the "imperative" semantics of JAVA, as described in Sections 2.2, 2.3, 2.4 and 2.5. Now the user can specify the things he wishes to prove, and subsequently (try to) prove it.

Suppose that we have the file `example.java`, as shown in Figure 4.3. Before we run the tool on it, we usually check whether it is accepted by the JAVA compiler by running `javac example.java`[7]. As expected, this does not report any errors. The next step is to generate

[6] At the moment, the tool generates output for PVS version 2.3 and ISABELLE99. It is planned that with new releases of these theorem provers, the tool will be kept up-to-date – if required.

[7] This is useful since, as explained above, the LOOP compiler only works on classes accepted by the JAVA compiler. Standardly, the compiler from the latest JDK version of Sun is used.

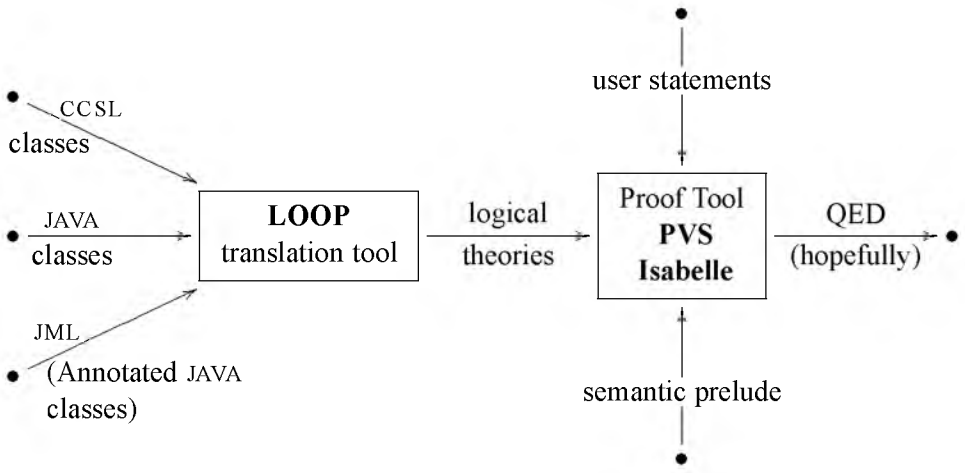

Figure 4.2: Using the LOOP tool

either PVS or ISABELLE theories. Since there are slight differences in the way to proceed in either case, both possibilities are described in some detail.

4.3.1 Using the LOOP tool and PVS

To generate PVS theories, the tool is run on the file `example.java` with the output type set to PVS: `run -pvs example.java`. This generates the following `.pvs` and `.prf` files:

```
A_basic                    B_basic
java_lang_Class_basic      java_lang_Exception_basic
java_lang_String_basic     java_lang_Throwable_basic
java_lang_Object_basic
```

The `.pvs` files contain the definitions and lemmas for each class, the `.prf` files contain the proofs of the lemmas. Notice that the implicit inheritance of `A` from `Object` is made explicitly by generating theories for class `Object` as well. Within the tool it is encoded which methods from `Object` should be translated. Most methods in `Object` deal with threads. At the moment we only deal with sequential JAVA, therefore these methods are ignored. The only methods of `Object` that are important for us are `equal`, `clone`, `toString` and the constructor. Class `Object` uses the class `String` (in the `toString` method), therefore this class is translated as well. Class `Class` is translated, because it provides useful methods, like the `instanceof` method, which are used very often. The other classes that are standard translated provide functionality *w.r.t.* exceptions. The theory (and file) names of the classes from the standard JAVA library, like `Object`, are extended with their package name, to avoid the generation of theories with the same name for classes in different packages.

Now PVS can be started. After loading the semantic prelude, the generated files are loaded and type checked. Notice that the user should guide PVS in which order to type check the files. Subsequently, the user can make his/her own PVS-file, say `B_user.pvs`, in which required properties about the JAVA classes can be stated (and proven)[8]. As explained above, typical

[8]The extension of the LOOP tool with JML annotations will also generate files containing proof obligations, say

– JAVA

```
class A {
  int i;
  void m() {
    i = 3;
  }
}

class B extends A {}
```

Figure 4.3: The contents of the file `example.java`

examples of user statements are termination results, class invariants, and requirements about the return value.

A typical proof about a method without loops and recursive calls proceeds as follows. First appropriate rewrite rules are loaded. These rewrite rules partly come from the semantic prelude, and partly are generated by the LOOP tool for all translated classes. Next the PVS proof command `REDUCE` is used, which applies as much rewriting as possible. If a method contains loops or recursive calls more elaborate proof techniques are required, *e.g.* using the Hoare logic rules as described in Chapter 5.

4.3.2 Using the LOOP tool and Isabelle

To generate ISABELLE theories, the tool is run with the output flag set to ISABELLE: `run -isa example.java`. Since in ISABELLE, each theory has its own file, this produces many files. For each class, eight theories are generated (at the moment). For each theory, a `.thy` and a `.ML` are generated, the first ones containing the definitions and axioms, the latter containing the theorems and rewrite sets, respectively. In this case, thus $7 \times 8 \times 2 = 112$ files are generated.

Now ISABELLE can be started and the generated files can be loaded and type checked, for example by making a user file `Example_user.thy` in which the appropriate rewrite theories are loaded. For each class `C`, among others, a theory `CRewrite` is generated, importing all the appropriate definitions describing the semantics of `C`, and containing all the appropriate rewrite rules. The file `Example_user.thy` imports all these rewrite theories.

– ISABELLE

```
Example_user = ARewrite +
               BRewrite +
               java_lang_ObjectRewrite
```

After loading `Example_user` the user can prove the required results, either by using automatic rewriting or by using other appropriate proof techniques. For each class `C` a set of appropriate rewrite rules `CRewrites` is generated. Also, for the definitions in the semantic prelude, a suitable set of rewrite rules (called `PreludeRewrites`) is available. These rewrite sets can be added to the simplification set in ISABELLE, and are then used in automatic rewriting.

`A_requirements.pvs` and `B_requirements.pvs`, stating proof obligations derived from the annotations. In that case, only the proofs remain to be done.

4.4 Some typical examples with automatic verification

To show the power of the translation via the LOOP tool, and the advantage of using a theorem prover for the verification, several example verifications are considered in this section. All these verifications could be done by automatic rewriting entirely. Later (*e.g.* in Chapter 5 and Chapter 7) verification examples will be discussed which need user interaction. The verifications that are discussed here, show several typical aspects of JAVA.

Evaluation order of arithmetic operators

The first topic that we discuss is the evaluation order. The evaluation order in JAVA is fixed, in contrast to *e.g.* C, where verification of expressions requires much more work (see [Nor99]). Consider for example the following JAVA class.

– JAVA
```
class Arithmetic {
  int m (int k) {
    int i = 0;
    return (k += i++ / i);
  }
}
```

It can be proven that the method `m` always terminates normally, returning the value of its parameter `k`. Notice that the fixed left-to-right-evaluation order ensures that no exception is thrown. Before the division by `i` is considered, `i` is increased by 1. Notice also that the correctness of this method is proven with respect to *all* parameters `k`. This is where (interactive) program verification differs from testing. In testing, this property can only be established for concrete values of `k`.

The verification of this method is done within PVS. After loading the appropriate theories, the following user statement is proven.

– PVS
```
ArithmeticUser : THEORY
BEGIN

  % code generated by the LOOP tool is loaded
  IMPORTING ...

  c : VAR [MemLoc? -> [OM? -> Arithmetic?IFace[OM?]]]
  p : VAR MemLoc?
  x : VAR OM?

  m_result : LEMMA
    ArithmeticAssert?(p)(c(p)) IMPLIES
      FORALL (k : int_java) :
        norm??(m?int(k)(c(p))(x)) AND
          res?[OM?, ExprAbn?[OM?], int_java]
               (m?int(k)(c(p))(x)) = k
END ArithmeticUser
```

This lemma states that for all possible value of `k` the method `m(k)` terminates normally, and its result will be equal to `k`. This proof takes about 42 rewrite steps, in about 60 seconds[9], of which about 3/4 is used for loading all the rewrite rules.

Late binding within a super call

The second verification deals with the following JAVA classes.

– JAVA

```
class C {
  void m() throws Exception { m(); }
}
class D extends C {
  void m() throws Exception {
    throw new Exception();
  }
  void test() throws Exception { super.m(); }
}
```

At a first glance, one might think that evaluation of the method `test` will not terminate. But in contrast, evaluation of method `test` will result in an exception. In the body of `test` the method `m` of `C` is called. This method calls `m` again, but – due to late binding (see Section 2.6) – this results in execution of `m` in `D`. However, calling `m` on an instance of class `C` directly will not terminate. The ISABELLE/HOL statement that have been proven is the following.

– ISABELLE

```
(* Code generated by the LOOP tool is loaded *)
Goal "DAssert'(p)(c(p)) ==> \
\     case DInterface.test' (c p) x of \
\       Hang'     => False\
\      |Norm' y   => False\
\      |Abnorm' a => True";
(* Simplifier *)
qed "m_in_D_Abnorm";
```

This lemma states that evaluation of `m` on an object with run-time type `D` will terminate abnormally. The proof of this lemma proceeds entirely by automatic rewriting again[10], after the generated rewrite rules are added to the simplifier. The crucial point in this verification is the binding of the extraction function for `super.m` on a `D` coalgebra d: OM $\rightarrow$ DIFace[OM] to the method body C_mbody(D2C(d)) (see Section 2.6.8).

It can also be proven that evaluation of `m` on an object with run-time type `C` will not terminate, *i.e.* will hang in our semantics[11].

[9] On a Pentium II, 266 MHz, with 96 MB RAM.

[10] On a Pentium II 266 Mhz with 96 MB RAM, running Linux, this takes about 71 sec, involving 5070 rewrite steps – including rewriting of conditions.

[11] To get this result, handling of recursive methods is necessary. In Section 2.6 we abstracted away from this. Basically, methods are described as least fixed points, iterated over hang.

– ISABELLE

```
(* Code generated by the LOOP tool is loaded *)
Goal "CAssert'(p)(c(p)) ==>\
\     case CInterface.m' (c p) x of \
\       Hang'     => True\
\      |Norm' y   => False\
\      |Abnorm' a => False";
(* Proof *)
qed "m_in_C_hangs";
```

The verification of this second lemma requires some more care, since it cannot be done via automatic rewriting (as this would loop). To prove non-termination, several unfoldings and the explicit introduction of an appropriate induction predicate are necessary.

Overriding and hiding

The next verification concerns the JAVA classes in Section 2.6.4, and establishes the properties mentioned there. For convenience we repeat the JAVA classes here.

– JAVA

```
class A {
  int i = 1;

  int m() { return i * 100; }

  int n() { return i +  m(); }
}

class B extends A {
  int i = 10;

  int m() { return i * 1000; }

  int test2() { return n(); }
}

class Test {
  int test1() {
    A[] ar = { new A(), new B() };
    return ar[0].i + ar[0].m() +
           ar[1].i + ar[1].m();
  }
}
```

Remember that – due to the dynamic binding of methods and the static binding of fields – `test1` returns 10102, and `test2` returns the value of `i` in `A` plus 1000 times the value of `i` in `B`. The PVS statements that have been proven are:

– PVS
```
% code generated by the LOOP tool is loaded
IMPORTING ...

test1_result : LEMMA
  TestAssert?(p)(c(p)) IMPLIES
    p < heap?top(x) IMPLIES
      norm??(test1?(c(p))(x)) AND
        res?(test1?(c(p))(x)) = 10102

test2_result : LEMMA
  BAssert?(p)(c(p)) IMPLIES
    p < heap?top(x) IMPLIES
      norm??(test2?(c(p))(x)) AND
        res?(test2?(c(p))(x)) =
          i(B?2?A(c(p)))(x) + i(c(p))(x) * 1000
```

The first lemma `test1` states that evaluation of `test1` terminates normally, returning `10102`. The second lemma states that evaluation of `test2` also terminates normally, and the return value equals the value of `i` from `A`, plus 1000 times the value of `i` from `B`.

The proofs of both lemmas proceed entirely by automatic rewriting[12]; the user only has to load the generated rewrite rules, and to start reducing. The functions CE2E and B2A play a crucial rôle in this verification. Hopefully the reader appreciates the semantic intricacies involved in the proof of the first lemma: array creation and access, local variables, object creation, implicit casting, and late binding.

Default initialisations

A typical aspect of JAVA is the immediate initialisation of (instance) fields with a default value. This allows a field to be used, before any value has been assigned to it explicitly. Consider for example the following JAVA classes.

– JAVA
```
class Example {}

class Initialise {

  Example e1;
  Example e2;

  Initialise () {
    e1 = e2;
    e2 = new Example ();
  }
}
```

[12]To give an impression, the proof of `test1` involves 790 rewrite steps, taking about 67 sec., on a 450 Mhz. Pentium III with 128 MB RAM under Linux.

In this example, if a new instance of class `Initialise` is created, the value of `e2` is assigned to `e1` before a value has been assigned to it. However, because of the default initialisation, this does not cause any problem, since reference values have a default initialisation to `null`. This behaviour is also incorporated in our semantics (see Section 2.6.11 for a more detailed explanation on the semantics of constructors), and it can be proven that each new instance of the class `Initialise` has two fields, `e1` and `e2`, where `e1` is `null` and `e2` is an instance of the class `Example`. This verification is done in ISABELLE/HOL.

– ISABELLE

```
(* Code generated by the LOOP tool is loaded *)
Goal "[|InitialiseAssert' p (c p);\
\       p < heap'top x |] ==>\
\       case new'Initialise constr'Initialise x of\
\         Hang' => False\
\        |Norm' y v => \
\           (case v of \
\              Null' => False\
\             |Reference' q => \
\                (case e1 (Initialise'clg \
\                            (get'type q y) q) y of\
\                   Null' => True\
\                  |Reference' r => False) &\
\                (case e2 (Initialise'clg \
\                            (get'type q y) q) y of\
\                   Null' => False\
\                  |Reference' r => get'type r y = \
\                                   ''Example''))\
\        |Abnorm' a => False";
(* Simplifier *)
qed "new'Initialise_result";
```

This lemma states that creation of a new instance of `Initialise` terminates normally, returning a reference to a new object. This object has two fields, `e1` and `e2`. The field `e1` is a null-pointer, the field `e2` points to an object which is an instance of class `Example`. The lemma again is proven by automatic rewriting[13].

4.5 Conclusions

This chapter discusses the use of the LOOP compiler in the verification of JAVA classes. The LOOP compiler works as a front-end tool for the theorem provers PVS and ISABELLE. It takes JAVA classes as input and generates appropriate PVS or ISABELLE theories, describing the semantics of the JAVA classes. Subsequently, properties of the JAVA classes can be verified in the

[13]The lemma is proven in approximately 55 seconds and 4330 rewrite steps (including almost 3000 failing attempts to rewrite the conditions of conditional rewrites).

theorem prover. In several examples, it is illustrated what kind of properties can be automatically verified.

We are not aware of other existing front-end tools, which translate JAVA classes (or other programming languages) into the input language of a theorem prover. There are several embeddings of programming languages in theorem provers, *e.g.* for C [Nor98] and JAVA [ON99], but in these cases the shift from program to specification for the theorem prover is always done by hand. Tool-supported verification of JAVA is achieved by the ESC static checker [DLNS98] and the Jive system [MPH00a]. The ESC static checker takes an annotated JAVA program and tries to check the annotations automatically. It cannot verify arbitrary properties, but it aims at preventing NullPointerExceptions, ArrayIndexOutOfBoundsExceptions and race conditions. The verifications are done statically and are quite fast. The Jive system allows the user to reason about a JAVA program using Hoare triples. The user selects which proof rules to apply (and gives an instantiation if necessary), and resulting proof obligations are passed on to PVS. The PVS system then tries to prove these proof obligations automatically. Within the Jive system, the user reasons at a syntactic level, in contrast to the LOOP approach, where reasoning is done at a semantic level. It is still too early to give a detailed comparison between the two approaches.

Chapter 5

A Hoare logic for Java

All the verifications of JAVA programs that are described so far, are done immediately in terms of the semantics as described in Chapter 2. But "[...] reasoning about correctness formulas in terms of semantics is not very convenient. A much more promising approach is to reason directly on the level of correctness formulas." (quote from [AO97, p. 57]). Hoare logic is a formalism for doing precisely this.

This chapter describes a concrete and detailed elaboration and adaptation of existing approaches to programming logics with exceptions, notably from [Chr84, Fok78, LP80, LS90, LvdS94, Lei95] (which are mostly in weakest precondition form). This elaboration and adaptation is done for a real-world programming language like JAVA. Although the basic ideas used here are well known, the elaboration is different. For example, in this elaboration there are many forms of abrupt termination, and not just one sole exception, and a semantics of statements and expressions as particular functions is used (as described in Chapter 2), and not a trace based semantics.

The logic presented here did not arise as a purely theoretical exercise, but was developed during actual verification of JAVA programs. The ability to handle abnormalities was crucially needed for the case studies described in Chapter 7, in particular when dealing with loops of which the bodies contain a `return` statements or throw an exception.

Hoare logic for a particular programming language consists of a series of deduction rules, involving constructs from the programming language, like assignment, if-then-else and composition (see Figure 5.1 below). In particular while loops have received much attention in Hoare logic, because they require a judicious and often non-trivial choice of a loop invariant. For more information, see *e.g.* [Bak80, Gri81, Apt81, Gor88, AO97]. There is a so-called "classical" body of Hoare logic, which applies to standard constructs from an idealised imperative programming language. This forms a well-developed part of the theory of Hoare logic. It is described in general terms, and not aimed at a particular programming language. In this chapter, an extension of standard Hoare logic is presented in which the different output options of statements and expressions results in different kinds of sentences (for e.g. Break or Return), see Section 5.3 below.

Gordon [Gor89] describes how the rules of Hoare logic are mechanically derived from the semantics of a simple imperative language. This enables both semantic and axiomatic reasoning about programs in this language. What we describe next may be seen as a deeper elaboration of this approach, building on ideas from [Chr84, LvdS94, Lei95]. All the proof rules that are presented in this chapter and in Appendix A are sound *w.r.t.* our semantics. Their correctness has been established in PVS and in ISABELLE. We did not consider completeness of the Hoare

logic.

It should be emphasised that the extension of Hoare logic that is introduced here applies only to a small (sequential, non-object-oriented) part of JAVA. Hoare logics for reasoning about concurrent programs may be found in [AO97], and for reasoning about object-oriented programs in [Boe99, AL97]. There is also more remotely related work on "Hoare logic with jumps", see [CH72, ACH76] (or also Chapter 10 by De Bruin in [Bak80]), but in those logics it is not always possible to reason about intermediate, "abnormal" states. In [PHM99] a programming logic for JAVA is described, which, in its current state, does not cover forms of abrupt termination – the focal point of this work. In [Ohe00] a sound and complete Hoare logic for JAVA is presented. This logic only deals with partial correctness. In this logic the predicates can distinguish whether a state is normal or abnormal, and for every language construct there is only one rule. In contrast, in the logic presented in this paper, there are many different rules per construct, for all possible termination modes.

This chapter is organised as follows. The first section briefly describes classical Hoare logic. Section 5.2 describes how this is tailored to JAVA. Then, Section 5.3 extends this to enable reasoning about abruptly terminating programs. Several proof rules, dealing with abrupt termination are discussed, including proof rules for loops as describes in Section 5.4. Section 5.5 describes Hoare logic rules for several of JAVA's more complicated programming constructs. An example verification is discussed in Section 5.6. The chapter ends with conclusions. Appendix A gives an overview of the rules of the logic.

5.1 Basics of Hoare logic

Traditionally, Hoare logic allows one to reason about simple imperative programs, containing assignments, conditional statements, block statements with local variables, while loops and for loops. It provides proof rules to derive the correctness of a complete program from the correctness of its constituents. Sentences (also called asserted programs) in this logic have the form $\{P\}\, S\, \{Q\}$, for partial correctness, or $[P]\, S\, [Q]$, for total correctness. They involve assertions P and Q in some logic (usually predicate logic), and statements S from the programming language that one wishes to reason about. The partial correctness sentence $\{P\}\, S\, \{Q\}$ expresses that if the assertion P holds in some state x and *if* the statement S, when evaluated in state x, terminates normally, resulting in a state x', then the assertion Q holds in x'. Total correctness $[P]\, S\, [Q]$ expresses something stronger, namely: if P holds in x, *then* S in x terminates normally, resulting in a state x' where Q holds. Figure 5.1 shows some well-known proof rules. In this figure the symbol ";" denotes statement composition, and the variable C is a Boolean condition. The predicate P in the `while` rule is often called the loop invariant.

Most classical partial correctness proof rules immediately carry over to total correctness. A well-known exception is the rule for the while statement, which needs an extra condition to prove termination. Consider for example the program (fragment) `while true do skip`. For every predicate P, it is easy to prove $[P]$ `skip` $[P]$. But the whole statement never terminates, so it should not be possible to conclude $[P]$ `while true do skip` $[P \wedge$ `false`$]$. An extra condition, which guarantees termination, should be added to the rule. The standard approach is to define a mapping from the underlying state space to some well-founded set and to require that whenever the body is executed, the result of this mapping decreases. As this can happen only finitely often, the loop has to terminate. Often this mapping is called the variant (in contrast to the loop invariant). This gives the following classical proof rule for total correctness of while

$$\frac{\{P\}\, S\, \{Q\} \qquad \{Q\}\, T\, \{R\}}{\{P\}\, S;\, T\, \{R\}}$$

$$\frac{\{P \wedge C\}\, S\, \{Q\} \qquad \{P \wedge \neg C\}\, T\, \{Q\}}{\{P\}\, \texttt{if}\ C\ \texttt{then}\ S\ \texttt{else}\ T\, \{Q\}}$$

$$\frac{\{P \wedge C\}\, S\, \{P\}}{\{P\}\, \texttt{while}\ C\ \texttt{do}\ S\, \{P \wedge \neg C\}}$$

Figure 5.1: Some proof rules of classical Hoare logic

statements.

$$\frac{[P \wedge C \wedge \text{variant} = n]\, S\, [P \wedge \text{variant} < n]}{[P]\, \texttt{while}\ C\ \texttt{do}\ S\, [P \wedge \neg C]}$$

5.1.1 Some limitations of Hoare logic

Hoare logic has had much influence on the way of thinking about (imperative) programming, but unfortunately it also has some shortcomings. First of all, it is not really feasible to verify non-trivial programs by hand. Most computer science students – at some stage during their training – have to verify some well-known algorithm, such as quicksort. At that moment they often decide never to do this again. One would like to have a tool, which applies many of the proof steps automatically, so that the user only has to interfere at crucial steps in the proof. Secondly, classical Hoare logic enables reasoning about program written in an ideal programming language, without side-effects, exceptions, abrupt termination of statements, *etc.* However, most widely-used (imperative) programming languages, including JAVA, do have side-effects, exceptions and the like.

The logic that is described here is especially tailored to JAVA(-like languages). Thus, it facilitates reasoning about programs containing *e.g.* side-effects, exceptions and abruptly terminating statements. The reasoning is done within a theorem prover (PVS or ISABELLE), and thus we are able to use the rewriting strategies of PVS and ISABELLE.

5.2 Hoare logic with normal termination

A first step in describing an appropriate Hoare logic for JAVA is to formalise the "traditional" notions of partial and total correctness, where only normal termination is considered. The predicates PartialNormal? and TotalNormal?, defined in Figure 5.2, formalise these notions in type theory, tailored to our JAVA semantics.

$$\begin{array}{l}\mathsf{pre, post: Self} \to \mathsf{bool, stat: Self} \to \mathsf{StatResult[Self]} \vdash \\ \quad \mathsf{PartialNormal?(pre, stat, post)} : \mathsf{bool} \stackrel{\text{def}}{=} \\ \quad\quad \forall x : \mathsf{Self.}\ \mathsf{pre}\, x \supset \ \mathsf{CASE\ stat}\, x\ \mathsf{OF}\ \{ \\ \quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{true} \\ \quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto \mathsf{post}\, y \\ \quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{true}\ \}\end{array}$$

$$\begin{array}{l}\mathsf{pre, post: Self} \to \mathsf{bool, stat: Self} \to \mathsf{StatResult[Self]} \vdash \\ \quad \mathsf{TotalNormal?(pre, stat, post)} : \mathsf{bool} \stackrel{\text{def}}{=} \\ \quad\quad \forall x : \mathsf{Self.}\ \mathsf{pre}\, x \supset \ \mathsf{CASE\ stat}\, x\ \mathsf{OF}\ \{ \\ \quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{false} \\ \quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto \mathsf{post}\, y \\ \quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{false}\ \}\end{array}$$

Figure 5.2: Definitions of partial and total correctness in type theory

It is easy to prove the validity of all the well-known Hoare logic proof rules, *e.g.* the skip axiom and the composition rule, using notations like $\{P\}\,[\![S]\!]\,\{Q\} = \mathsf{PartialNormal?}(P, [\![S]\!], Q)$. Notice that these proof rules are given at a semantic level, in contrast to traditional Hoare logics, which work syntactly, directly on the source code. In our approach, the source code is translated first into a corresponding type-theoretic term, and subsequently the Hoare logic rules are applied to this term. But since the translation from JAVA source code to the type-theoretic description is compositional, there is not much difference: during a proof one can still follow the program structure of the original program. However, the advantage of working on a semantic level is that we are able to construct rules for *e.g.* CATCH-STAT-RETURN, which is implicit in the syntax, but explicit in the semantics.

$$\{P\}\ \mathsf{skip}\ \{P\}$$

$$\frac{\{P\}\ S\ \{Q\} \qquad \{Q\}\ T\ \{R\}}{\{P\}\ S\,;\ T\ \{R\}}$$

More over, it is easy to incorporate side-effects into these rules. For example, the following proof rule for the conditional statement is proven [1].

[1]The use of the (translated) JAVA condition C in the if-then-else rule, and also in other rules below, is deliberately sloppy, for readability. This C is a Boolean expression, of type Self→ ExprResult[Self, bool], but occurs in $P \wedge C$, where P is a predicate Self→bool. The latter conjunction $\wedge$ in a state x : Self should be understood as: $P\,x$, and $C\,x$ terminates normally, and its result is true.

– TYPE THEORY

$$\frac{\{P \wedge C\}\ \mathsf{E2S}(C)\ ;\ S\ \{Q\} \qquad \{P \wedge \neg C\}\ \mathsf{E2S}(C)\ ;\ T\ \{Q\}}{\{P\}\ \mathsf{IF\text{-}THEN\text{-}ELSE}(C)(S)(T)\ \{Q\}}$$

The classical side-effect-free rule is a special case of this rule.

Similarly, the following proof rule for total correctness of the while statement can be proven (where we assume that $<$ is some well-founded order).

– TYPE THEORY

$$\frac{\begin{array}{c}[P]\ \mathsf{E2S}(C)\ [\mathsf{true}] \\ \forall n[P \wedge C \wedge \mathit{variant} = n]\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(\mathit{ll})(S)\ [P \wedge \mathit{variant} < n] \\ \{P \wedge \neg C\}\ \mathsf{E2S}(C)\ \{Q\}\end{array}}{[P]\ \mathsf{WHILE}(\mathit{ll})(C)(S)\ [Q]}$$

Recall from Section 2.4.3 that $\mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(\mathit{ll})(S)$ is called the iteration body of the loop. To prove total correctness of the `while` statement, the following has to be shown: (1) evaluation of the condition always terminates normally, (2) if the condition evaluates to true, the iteration body terminates normally, preserving the invariant P and with some (well-founded) variant decreasing, and (3) if the conditions evaluates to false, the postcondition should be established. The difference with the traditional `while` rule comes from the fact that expressions in JAVA can have side-effects and throw exceptions.

Also extra proof rules, capturing the correctness of abruptly terminating statements, can be formulated (and proven). As an example, the following rule states that given a labeled block, containing some statement S, followed by an appropriately labeled `break` statement, it suffices to look at the correctness of S.

– TYPE THEORY

$$\frac{[P]\ S\ [Q]}{[P]\ \mathsf{CATCH\text{-}BREAK}(l)(S\ ;\ \mathsf{BREAK\text{-}LABEL}(l))\ [Q]}$$

For other abnormalities similar rules can be formulated immediately.

For expressions, a similar notion of partial and total correctness is defined. However, there is one important difference: the postcondition is a predicate over the (result) state and the return value, thus allowing to use the return value in the postcondition. Hoare sentences over expressions with result type Outhave a post-condition with type Self $\rightarrow$ Out $\rightarrow$ bool. Total correctness over expressions is defined as follows.

– TYPE THEORY

$$\begin{array}{l}\mathsf{pre}, \mathsf{post}: \mathsf{Self} \rightarrow \mathsf{bool}, \mathsf{expr}: \mathsf{Self} \rightarrow \mathsf{ExprResult[Self, Out]}\ \vdash \\ \quad \mathsf{TotalNormal?(pre, expr, post)}\ :\ \mathsf{bool} \stackrel{\mathrm{def}}{=} \\ \quad\quad \forall x : \mathsf{Self}.\ \mathsf{pre}\, x \supset\ \mathsf{CASE}\ \mathsf{expr}\, x\ \mathsf{OF}\ \{ \\ \quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{false} \\ \quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto \mathsf{post}\ (y.\mathsf{ns})\ (y.\mathsf{res}) \\ \quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{false}\ \}\end{array}$$

A similar definition is given for partial correctness over expressions.

5.3 Hoare logic with abrupt termination

Unfortunately, the proof rules for normal termination are not sufficient for reasoning about arbitrary JAVA programs. To achieve this, it is necessary to have a "correctness notion" of being in an abnormal state, *e.g.* if execution of S starts in a state satisfying P, then execution of S terminates abruptly, because of a `return`, in a state satisfying Q. To this end, the notions of abnormal correctness are introduced. They appear in four forms, corresponding to the four possible kinds of abnormalities. Rules are formulated to derive the (abnormal) correctness of a program compositionally. These rules allow the user to move back and forth between the various correctness notions.

The first notion of abnormal correctness that is introduced is partial break correctness (with notation: $\{P\}\, S\, \{\mathsf{break}(Q, l)\}$), meaning that if execution of S starts in some state satisfying P, and execution of S terminates in an abnormal state, because of a `break`, then the resulting abnormal state satisfies Q. If the `break` is labeled with `lab`, then $l = \mathsf{up}(\text{"}\texttt{lab}\text{"})$, otherwise $l = \mathsf{bot}$.

Naturally, there exists also total break correctness ($[P]\, S\, [\mathsf{break}(Q, l)]$), meaning that if execution of S starts in some state satisfying P, then execution of S terminates in an abnormal state, satisfying Q, because of a `break`. If this `break` is labeled with a label `lab`, then $l = \mathsf{up}(\text{"}\texttt{lab}\text{"})$, otherwise $l = \mathsf{bot}$. Continuing in this manner leads to the following eight notions of abnormal correctness.

partial break correctness	$\{P\}\, S\, \{\mathsf{break}(Q, l)\}$
partial continue correctness	$\{P\}\, S\, \{\mathsf{continue}(Q, l)\}$
partial return correctness	$\{P\}\, S\, \{\mathsf{return}(Q)\}$
partial exception correctness	$\{P\}\, S\, \{\mathsf{exception}(Q, e)\}$
total break correctness	$[P]\, S\, [\mathsf{break}(Q, l)]$
total continue correctness	$[P]\, S\, [\mathsf{continue}(Q, l)]$
total return correctness	$[P]\, S\, [\mathsf{return}(Q)]$
total exception correctness	$[P]\, S\, [\mathsf{exception}(Q, e)]$

For expressions, we get similar notions of partial and total exception correctness.

It is tempting to change the standard notation $\{P\}\, S\, \{Q\}$ and $[P]\, S\, [Q]$ into $\{P\}\, S\, \{\mathsf{norm}(Q)\}$ and $[P]\, S\, [\mathsf{norm}(Q)]$ to bring it in line with the new notation, but we stick to the standard notation for normal termination.

The formalisation of these correctness notions in type theory is straightforward. As an example, consider the predicates PartialReturn? and TotalBreak? for partial return and total break correctness. They are used to give meaning to the notations $\{P\}\, [\![S]\!]\, \{\mathsf{return}(Q)\} = \mathsf{PartialReturn?}(P, [\![S]\!], Q)$ and $[P]\, [\![S]\!]\, [\mathsf{break}(Q, l)] = \mathsf{TotalBreak?}(l)(P, [\![S]\!], Q)$. These predicates are defined in Figure 5.3.

The predicate expressing partial and total exception correctness have a slightly different definition, because their postconditions depend on the result state and on the occurred exception, thus having type $\mathsf{Self} \to \mathsf{RefType} \to \mathsf{bool}$.

Many straightforward proof rules can be formulated and proven for these correctness notions. First of all, there are the analogues of the skip axiom.

$$\begin{array}{l}
\mathsf{pre, post}: \mathsf{Self} \to \mathsf{bool}, \mathsf{stat}: \mathsf{Self} \to \mathsf{StatResult[Self]} \vdash \\
\quad \mathsf{PartialReturn?(pre, stat, post)} : \mathsf{bool} \stackrel{\mathrm{def}}{=} \\
\quad\quad \forall x : \mathsf{Self}.\, \mathsf{pre}\, x \supset \mathsf{CASE\ stat}\, x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{true} \\
\quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto \mathsf{true} \\
\quad\quad\quad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{CASE}\, a\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{excp}\, e \mapsto \mathsf{true} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{rtrn}\, z \mapsto \mathsf{post}\, z \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{break}\, b \mapsto \mathsf{true} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{cont}\, c \mapsto \mathsf{true}\ \}\ \}
\end{array}$$

$$\begin{array}{l}
l : \mathsf{lift[string]}, \mathsf{pre, post}: \mathsf{Self} \to \mathsf{bool}, \mathsf{stat}: \mathsf{Self} \to \mathsf{StatResult[Self]} \vdash \\
\quad \mathsf{TotalBreak?}(l)(\mathsf{pre, stat, post}) : \mathsf{bool} \stackrel{\mathrm{def}}{=} \\
\quad\quad \forall x : \mathsf{Self}.\, \mathsf{pre}\, x \supset \mathsf{CASE\ stat}\, x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{false} \\
\quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto \mathsf{false} \\
\quad\quad\quad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{CASE}\, a\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{excp}\, e \mapsto \mathsf{false} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{rtrn}\, z \mapsto \mathsf{false} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{break}\, b \mapsto b.\mathsf{blab} = l \wedge \mathsf{post}(b.\mathsf{bs}) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{cont}\, c \mapsto \mathsf{false}\ \}\ \}
\end{array}$$

Figure 5.3: Definitions of partial return correctness and total break correctness in type theory

$$\{P\}\ \mathsf{RETURN}\ \{\mathsf{return}(P)\}$$

Then there are rules, expressing how these correctness notions behave with "traditional" program constructs, such as statement composition. Notice that these rules are always about one correctness notion.

$$\frac{[P]\ S\ [\mathsf{return}(R)]}{[P]\ S\,;\ T\ [\mathsf{return}(R)]}$$

$$\frac{[P]\ S\ [Q] \qquad [Q]\ T\ [\mathsf{return}(R)]}{[P]\ S\,;\ T\ [\mathsf{return}(R)]}$$

$$\frac{\{P\}\ S\ \{\mathsf{return}(R)\} \qquad \{P\}\ S\ \{Q\} \qquad \{Q\}\ T\ \{\mathsf{return}(R)\}}{\{P\}\ S\,;\ T\ \{\mathsf{return}(R)\}}$$

To prove total return correctness of statement composition, either the first statement should terminate abruptly, because of return, or it should terminate normally, and the second statement should terminate abruptly. These two possibilities are expressed by the first two proof rules. The last proof rule is concerned with partial return correctness. It is assumed that the statement composition terminates abruptly, because of a return. There are two possibilities: either the first statement terminates abruptly, or the second statement produces the abnormally. Both cases have to be considered. Notice that in reasoning about total correctness, the choice of the proof rule reflects where the abnormality occurred, while in reasoning about partial correctness all possibilities have to be considered.

Finally, there are rules to move between two correctness notions, from normal to abnormal and vice versa. Here some examples for the return statement again.

$$\frac{\{P\}\ S\ \{\mathsf{return}(Q)\} \qquad \{P\}\ S\ \{Q\}}{\{P\}\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(S)\ \{Q\}}$$

$$\frac{[P]\ S\ [\mathsf{return}(Q)]}{[P]\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(S)\ [Q]}$$

$$\frac{[P]\ S\ [Q]}{[P]\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(S)\ [Q]}$$

$$\frac{[P]\ S\ [\mathsf{return}(\lambda x : \mathsf{Self}.\ R\,x\,(v\,x))]}{[P]\ \mathsf{CATCH\text{-}EXPR\text{-}RETURN}(S)(v)\ [R]}$$

The first rule states that to show partial correctness of CATCH-STAT-RETURN(S) both partial correctness and partial return correctness of S have to be shown. This can be understood as follows: partial correctness of CATCH-STAT-RETURN(S) assumes normal termination of CATCH-STAT-RETURN(S). Looking at the definition of CATCH-STAT-RETURN, it follows that either S terminates normally, or it produces a return abnormality. In both cases, the postcondition has to be established by S. To show total correctness of CATCH-STAT-RETURN(S), there are two rules that can applied. To show normal termination of CATCH-STAT-RETURN(S) it suffices to show that S terminates abruptly, because of a return, or that S terminates normally. These two possibilities are captured by the second and third proof rule. Finally, the last rule states that total correctness of CATCH-EXPR-RETURN(S)(v) follows from total return correctness of S. Notice that in this rule the postcondition Q has type Self $\rightarrow$ Out $\rightarrow$ bool. To transform this into a postcondition of type Self $\rightarrow$ bool, Q is applied to $v\,x$, which is the result value of CATCH-EXPR-RETURN(S)(v).

Most of these proof rules are easy and straightforward to formulate, but proof rules for while loops with abrupt termination are more difficult to formulate. This is described in the next section.

5.4 Hoare logic of while loops with abrupt termination

Recall that in classical Hoare logic, reasoning about while loops involves the following ingredients: (1) an invariant, *i.e.* a predicate over the state space which is true initially and after each iteration of the while loop; (2) a condition, which is false after normal termination of the while loop; (3) a body, whose execution is iterated a number of times; (4) (when dealing with total correctness) a variant, *i.e.* a mapping from the state space to some well-founded set, which strictly decreases every time the body is executed.

To see what is needed to extend this to abnormal correctness, first a silly example of an abruptly terminating while loop is discussed.

– JAVA
```
while (true) { if (i < 10) { i++; }
               else { break; } }
```

This loop always terminates, and a variant can be constructed to show this, but after termination it cannot be concluded that the condition has become false. But by inspecting the code we see that $i \geq 10$ must have caused termination of the loop. After termination of the loop, we want to be able to use this information. Thus proof rules have to be formulated in such a way that, in this case, it can be concluded that after termination of the while loop $i < 10$ does not hold (anymore). This desire leads to the development of special rules for partial and total abnormal correctness of while loops. Below, the partial and total break correctness rules are described in full detail. The rules for the other abnormalities are basically the same.

5.4.1 Partial break while rule

Suppose that we have a while loop WHILE(I_1)(C)(S), which is executed in a state satisfying P. We wish to prove that if the while loop terminates abruptly, because of a break, then the result

state satisfies Q – where P is the loop invariant and Q is the predicate that holds upon abrupt termination (in the example above: $i \geq 10$). A natural condition for the proof rule is thus that if the body terminates abruptly, because of a break, then Q should hold. Furthermore, we have to show that P is an invariant if the body terminates normally.

$$\frac{\{P\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(l_1)(S)\ \{P\} \qquad \{P\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(l_1)(S)\ \{\mathsf{break}(Q, l_2)\}}{\{P\}\ \mathsf{WHILE}(l_1)(C)(S)\ \{\mathsf{break}(Q, l_2)\}}$$

Thus, assume: (1) if the iteration body $\mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(l_1)(S)$ is executed in a state satisfying P and terminates normally, then P still holds, and (2) if the iteration body is executed in a state satisfying P and ends in an abnormal state, because of a break, then this state satisfies some property Q. Then, if the while statement is executed in a state satisfying P and it terminates abruptly, because of a break, then its final state satisfies Q.

Soundness of this rule is easy to see (and to prove): suppose we have a state satisfying P, in which $\mathsf{WHILE}(l_1)(C)(S)$ terminates abruptly, because of a break. This means that the iterated statement $\mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(l_1)(S)$ terminates normally a number of times. All these times, P remains true. However, at some stage the iterated statement must terminate abruptly, because of a break, labeled l_2, and then the resulting state satisfies Q. As this is also the final state of the whole loop, we get $\{P\}\ \mathsf{WHILE}(l_1)(C)(S)\ \{\mathsf{break}(Q, l_2)\}$

5.4.2 Total break while rule

Next a proof rule for the total break correctness of the while statement is presented. Suppose there exists a state satisfying $P \wedge C$ and it has to be proven that execution of $\mathsf{WHILE}(l_1)(C)(S)$ in this state terminates abruptly, because of a break, resulting in a state satisfying Q. It has to be shown that (1) the iteration body terminates normally only a finite number of times (using a variant), and (2) if the iteration body does not terminate normally, it must be because of a break, resulting in an abnormal state, satisfying Q. This gives (assuming that $<$ is a well-founded order):

– TYPE THEORY

$$\frac{[P]\ \mathsf{CATCH\text{-}BREAK}(l_2)(\mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(l_1)(S))\ [\mathsf{true}] \qquad \begin{array}{l}\forall n.\ \{P\ \wedge\ C\ \wedge\ \mathit{variant} = n\} \\ \quad \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(l_1)(S) \\ \quad \{P\ \wedge\ C\ \wedge\ \mathit{variant} < n\}\end{array} \qquad \{P\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(l_1)(S)\ \{\mathsf{break}(Q, l_2)\}}{[P]\ \mathsf{WHILE}(l_1)(C)(S)\ [\mathsf{break}(Q, l_2)]}$$

The first condition states that execution of the iteration body followed by a CATCH-BREAK, in a state satisfying $P \wedge C$, always terminates normally. Thus the iteration body itself must terminate either normally, or abruptly because of a break. The second condition expresses that if the iteration body terminates normally, the invariant and condition remain true and some variant decreases. Thus, the iteration body can only terminate normally a finite number of times. Finally, the last condition of this rule states that when the iteration body terminates

abruptly (because of a break), the resulting state satisfies Q. Soundness of this rule is easy to prove.

In [Chr84] a comparable rule "(R9)" is presented, which is slightly more restrictive: it requires that the abnormality occurs when the variant becomes 0. In our case it is only required that it should occur, but it is not specified when.

5.5 More Hoare logic for Java

The statements for which Hoare logic sentences have been discussed so far are the typical statements of a simple while language. This section describes Hoare logic rules for more complicated language constructs, such as block statements (introducing local variables), array operations and (possibly qualified) method calls. This presentation is mainly based on [Apt81], which presents proof rules for these language constructs and discusses their soundness and completeness. In this section, it is discussed how these rules are adapted to JAVA, and how abrupt termination is incorporated. This section is structured according to [Apt81], first discussing block statements, then array operations and finally method calls. We do not consider parameterless method calls separately.

5.5.1 Block statements and local variables

The first language extension for which Hoare logic proof rules are considered are block statements, which introduce local variables. Remember that, as explained in Section 2.6.8, the LET construct is used to represent JAVA's local variables in type theory. In a LET expression, appropriate get-operations (for access) and put-operations (for assignment) on the stack are linked to the local variables in that block. For example, a JAVA program fragment `{int i; S}`, where `S` is some arbitrary JAVA statement, is translated into the following fragment in type theory (for a particular cell location c, which is determined by the LOOP compiler).

– TYPE THEORY ———————————————————————————————

$$\begin{array}{l}\mathsf{LET\ i} = \mathsf{get\_int}(\mathsf{stack}(\mathsf{ml} = \mathsf{stacktop}\, x, \mathsf{cl} = c)) \\ \quad\ \ \mathsf{i\_becomes} = \mathsf{get\_int}(\mathsf{stack}(\mathsf{ml} = \mathsf{stacktop}\, x, \mathsf{cl} = c)) \\ \mathsf{IN}\ [\![\mathtt{S}]\!]\end{array}$$

———

All free occurrences of `i` in `S` are bound by the LET statement. A way to view this is to consider $[\![\mathtt{S}]\!]$ to be of type $(\mathsf{Self} \to \mathsf{int}) \times (\mathsf{Self} \to \mathsf{int} \to \mathsf{Self}) \to \mathsf{Self} \to \mathsf{StatResult[Self]}$, thus as a function which is parametrised with the access and assignment operations for the local variables.

In [Apt81], the following rule is presented for block statements (written in JAVA syntax, where ω is a symbol meaning "undefined", and the variable `x` is declared to be of some type `T`[2]).

$$\frac{\{\lambda z : \mathsf{OM}.\ P\, z \wedge y = \omega\}\ \mathtt{S}[y/\mathtt{x}]\ \{Q\}}{\{P\}\ \{\mathtt{T\ x;\ S}\}\ \{Q\}} \text{ where } y \text{ not free in } P,\ S \text{ and } Q$$

[2] In [Apt81] the rule is presented in untyped form. The type `T` can be both a primitive type and a reference type.

In this rule, x is renamed to y to avoid possible name clashes. The expression $y = \omega$ captures the idea of initialisation. The effect of this rule is that the local variable is moved from the program to the assertions.

To adapt this to our setting, some adaptations have to be made, because we have two functions (one for access, one for assignment) which together represent the local variable. Instead of a new free variable, we get a new cell location on the stack, in which the local variable is stored. This leads to the following proof rule (in type-theoretic "syntax"), where again the ω symbol is used for default initialisation.

– TYPE THEORY

$$\frac{\begin{array}{l}\{\lambda z : \mathsf{OM}.\ P\,z \ \wedge\ \mathsf{get\_typ}(\mathsf{stack}(\mathsf{ml} = ml, \mathsf{cl} = c))\,z = \omega\} \\ \quad S\ (\mathsf{get\_typ}(\mathsf{stack}(\mathsf{ml} = ml, \mathsf{cl} = c)), \\ \qquad \mathsf{put\_typ}(\mathsf{stack}(\mathsf{ml} = ml, \mathsf{cl} = c))) \\ \{Q\}\end{array}}{\begin{array}{l}\{P\} \\ \quad \mathsf{LET}\ y = \mathsf{get\_typ}(\mathsf{stack}(\mathsf{ml} = ml, \mathsf{cl} = cl)), \\ \qquad y\_becomes = \mathsf{put\_typ}(\mathsf{stack}(\mathsf{ml} = ml, \mathsf{cl} = cl)) \\ \quad \mathsf{IN}\ S(y, y\_becomes) \\ \{Q\}\end{array}}$$

This rule can be reformulated with the names of the local variables bound to the locations in the assertions. This has the advantage that the names of the local variables can be used in the assertions, and it is not necessary to use their locations.

– TYPE THEORY

$$\frac{\begin{array}{l}\forall y : \mathsf{Self} \to \mathsf{Out}.\ \forall y\_becomes : \mathsf{Self} \to \mathsf{Out} \to \mathsf{Self}. \\ \quad \{\lambda z : \mathsf{Self}.\ \ P\,z \wedge \\ \qquad\qquad y = \mathsf{get\_typ}(\mathsf{stack}(\mathsf{ml} = ml, \mathsf{cl} = cl)) \wedge \\ \qquad\qquad y\_becomes = \mathsf{put\_typ}(\mathsf{stack}(\mathsf{ml} = ml, \mathsf{cl} = cl)) \wedge \\ \qquad\qquad y\,z = \omega\} \\ \quad S(y, y\_becomes) \\ \quad \{Q\}\end{array}}{\begin{array}{l}\{P\} \\ \quad \mathsf{LET}\ y = \mathsf{get\_typ}(\mathsf{stack}(\mathsf{ml} = ml, \mathsf{cl} = cl)), \\ \qquad y\_becomes = \mathsf{put\_typ}(\mathsf{stack}(\mathsf{ml} = ml, \mathsf{cl} = cl)) \\ \quad \mathsf{IN}\ S(y, y\_becomes) \\ \{Q\}\end{array}}$$

Similar rules hold for total correctness and all kinds of abnormal correctness. Return variables and parameters are treated in the same way as local variables. To use these rules, special versions of the translated method bodies are required, which are parametrised over the local variables. These special bodies can be generated with a special compiler flag.

5.5.2 Array operations

The following program constructs for which Hoare logic rules are discussed are array operations. A well-known problem in stating Hoare logic rules for array assignments is that an

assignment `a[i] = t` also can have an effect on the value of `i`. For example, suppose that `a` is an array of integers, containing the value `2` at all positions. After the assignment `a[a[2]] = 1`, it should not be possible to prove that `a[a[2]]` equals `1`, since `a[2]` evaluates to `1` and `a[1]` still equals `2`. Thus the proof rules for normal assignments cannot be immediately reused for assignments on arrays.

The solution that is proposed in [Apt81] is to adapt the definition of substitution. For simple array index expressions the normal definition of substitution is still used, but complex array index (like in `a[a[2]]`) expressions are first "quantified out", *i.e.* rewritten into an expression containing only simple index expressions, and substitution is applied on the resulting expression. For example, the expression $\texttt{a[a[2]]} = 1$ becomes $\exists z.(\texttt{a}[z] = 1 \wedge z = \texttt{a[2]})$. Substitution over this expression simplifies as follows.

$$
\begin{array}{ll}
 & (\texttt{a[a[2]]} = 1)[t/\texttt{a[s]}] \\
\equiv & \{\text{"quantified out" assertion}\} \\
 & \exists z.(\texttt{a}[z] = 1 \wedge z = \texttt{a[2]})[t/\texttt{a[s]}] \\
\equiv & \{\text{definition of substitution}\} \\
 & \exists z.((\mathsf{IF}\ z = \texttt{s}\ \mathsf{THEN}\ t\ \mathsf{ELSE}\ \texttt{a}[z]) = 1 \wedge \\
 & \quad (\mathsf{IF}\ 2 = \texttt{s}\ \mathsf{THEN}\ t\ \mathsf{ELSE}\ \texttt{a[2]}) = z)
\end{array}
$$

Thus, new variables are introduced which remember the old value of the index expression. Defining substitution over array index expressions using this "quantifying out" method, the following proof rule can be proven for array assignments.

$$\{P[t/\texttt{a[s]}]\}\ \texttt{a[s]} = \texttt{t}\ \{P\}$$

Using this rule, and the substitution as explained above, we find that in order to prove that `a[a[2]] = 1` is the postcondition for the assignment `a[a[2]] = 1`, the precondition has to imply that

$$\exists z.((\mathsf{IF}\ z = \texttt{a[2]}\ \mathsf{THEN}\ 1\ \mathsf{ELSE}\ \texttt{a}[z]) = 1 \wedge (\mathsf{IF}\ 2 = \texttt{a[2]}\ \mathsf{THEN}\ 1\ \mathsf{ELSE}\ \texttt{a[2]}) = z)$$

which follows from $\texttt{a[2]} \neq 2 \vee \texttt{a[1]} = 1$. Thus, if all the elements in array `a` have the value 2, the postcondition cannot be established.

To adapt this rule to our JAVA semantics, it has to be taken into account that evaluation of the array, index and data expressions can have side-effects and that exceptions can be thrown. These considerations lead to the following partial correctness proof rule for assignments to an array of objects (*i.e.* ref_assign_at).

– TYPE THEORY ———————————————————————————————

$$
\frac{\begin{array}{l}
\exists r\colon \mathsf{MemLoc}.\ \exists i\colon \mathsf{int}. \\
\quad \{P\}\ \mathit{array\_expr}\ \{\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge \mathsf{CASE}\ v\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{null} \mapsto \mathsf{false} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{ref}\ p \mapsto p = r\}\} \\
\quad \{R\}\ \mathit{index\_expr}\ \{\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{int}.\ S\,x \wedge v = i\} \\
\quad \{S\}\ \mathit{data\_expr}\ \{\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{RefType}.\ Q(\mathsf{put\_ref}(\mathsf{heap}(\mathsf{ml} = r, \mathsf{cl} = i))\,x(v))(v)\}
\end{array}}{\{P\}\ \mathsf{ref\_assign\_at}(\mathit{array\_expr}, \mathit{index\_expr})(\mathit{data\_expr})\ \{Q\}}
$$

———

This proof rule should be read as follows. Suppose that an array assignment is evaluated in a

state satisfying P, terminating normally. We wish to show that after termination Q holds. First, *array_expr* is evaluated, resulting in an intermediate state, satisfying some predicate R. Also, *array_expr* returns a non-null reference to some location p (otherwise ref_assign_at would have produced an exception). Next, the *index_expr* is evaluated in this intermediate state satisfying R, returning a state satisfying S and an index value. Notice that the values of the reference and the index expression are remembered in the logical variables r and i, so that they can be used later, thus avoiding the problem with side-effects on the various expressions. The index is known to be in between the array bounds, otherwise an exception would have been thrown by ref_assign_at. Then, the *data_expr* is evaluated. The state that is produced by this evaluation should satisfy Q *after* writing the data value in the array at the appropriate position. Thus, it can be concluded that after the array assignment operation Q holds.

This rule seems to be very different from the rule in [Apt81], but actually it is not. The postcondition of *data_expr* is the precondition to the real assignment operation, and it basically states that Q[`a[i]/t`] should be true.

However, there is a problem when one wishes to use this rule, because the values of r and i have to be instantiated before the state is known. Often the values for these variables will depend on the state space, *e.g.* to prove the correctness of the assignment `a[a[2]] = 1`, i will equal `a[2]`, which clearly depends on the current state. Therefore, an alternative form of the definition is given, where the logical variables r and i actually are parametrised over the state space. To be able to use this rule, one has to show that the evaluation of *index_expr* does not affect the value of r. This gives the following alternative proof rule for ref_assign_at.

– TYPE THEORY ———————————————————————————————

$$\frac{\begin{array}{l}\exists r\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \exists i\colon \mathsf{OM} \to \mathsf{int}.\ \forall z\colon \mathsf{OM}.\ \forall w\colon \mathsf{OM}.\\ \{P\}\mathit{array\_expr}\{\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{RefType}.\ \ R\,x\ \wedge\ \mathsf{CASE}\ v\ \mathsf{OF}\ \{\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{null} \mapsto \mathsf{false}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{ref}\,r \mapsto r = r\,x\}\}\\ \{\lambda x\colon \mathsf{OM}.\ R\,x\ \wedge\ x = z\}\mathit{index\_expr}\{\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{int}.\ S\,x\ \wedge\ v = i\ \wedge\ r\,x = r\,z\}\\ \{\lambda x\colon \mathsf{OM}.\ S\,x\ \wedge\ x = w\}\\ \quad \mathit{data\_expr}\\ \{\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{RefType}.\ Q(\mathsf{put\_ref}(\mathsf{heap}(\mathsf{ml} = r\,w, \mathsf{cl} = i\,w))\,x(v))(v)\}\end{array}}{\{P\}\ \mathsf{ref\_assign\_at}(\mathit{array\_expr}, \mathit{index\_expr})(\mathit{data\_expr})\ \{Q\}}$$

Using this rule, we can prove for example

$$\{[\![\texttt{a[2]}]\!] \neq 2\ \vee\ [\![\texttt{a[1]}]\!] = 1\}\,[\![\texttt{a[a[2]] = 1}]\!]\,\{[\![\texttt{a[a[2]]}]\!] = 1\}$$

In a similar way rules can be formulated for other array operations (assignment to a primitive array, array access), total correctness of array operations, and exception correctness of array operations. In a proof rule for total correctness, the assumptions require that it is shown that the array reference is non-null, the index-value is between bounds and the run-time type of the *data_expr* is assignable to the array. Thus, to use the total correctness rule for array assignment, these properties have to be shown by the user.

Since all array operations are expressions, the only case of abrupt termination that has to be considered is because of exceptions. Several proof rules can be formulated, which describe the possible sources of exceptions in array operations.

5.5.3 Non recursive method calls

The last language construct for which proof rules are discussed in this section are method calls. As in the rest of this thesis, only non-recursive method calls are considered. For recursive method calls, appropriate proof rules can be formulated and proven as well, but this falls out of the scope of this thesis. JAVA has a call-by-value parameter mechanism, so this is the only case that we consider here.

In the discussion of proof rules for non-recursive method calls, Apt [Apt81] first defines the meaning of method calls as follows (adapted to JAVA syntax). Given a method `m(A x){S;}` with some arbitrary body `S`, the following notation is introduced.

$$\texttt{mbody(t)} \equiv \texttt{\{A u; u = t; S[u/x];\}}$$

where `u` is not free in `S`, `x` and `t`. The meaning of a method call is now defined as follows.

$$[\![\texttt{m(t)}]\!] \stackrel{\text{def}}{=} [\![\texttt{mbody(t)}]\!]$$

Notice that this is a simplified version of the translated method bodies as presented in Section 2.6.8 (transforming the local variables into a LET expression). For convenience we wrap the method body up in only one LET, but this is basically the same.

Using this definition, the following proof rule can be proven.

$$\frac{\{P\}\,\texttt{mbody(x)}\,\{Q\}}{\{P\}\,\texttt{m(x)}\,\{Q\}}$$

Adapting this to our context gives the following proof rule.

– TYPE THEORY –

$$\frac{\forall x : \mathsf{Self}.\ \mathsf{m}(c\ p)\ x = \mathsf{mbody}(d\ p)(sc\ p)(p)\ x \qquad \{P\}\,\mathsf{mbody}(d\ p)(sc\ p)(p)\,\{Q\}}{\{P\}\,\mathsf{m}(c\ p)\,\{Q\}}$$

Notice that this rule does not deal with late binding; it only enables replacement of a method call with a method body if it is clear which method body is selected. The first assumption relates the method call to the method body. It is supposed to be implied by the Assert predicate of the class implementing `m`. Notice that m and mbody can be applied to different coalgebras ($c\ p$ and $d\ p$, respectively), so the implementation of m can have been found in a superclass.

The second assumption states that normal termination of the method bodies results in a state satisfying Q. From this, it can be concluded that normal termination of the method call also results in a state satisfying Q.

Again, many variations to this rule are possible, *e.g.* non-void methods (*i.e.* expressions), parametrised methods, total correctness and exception correctness. However, all these rules are not significantly different from this one. Notice that the other kinds of abnormalities do not have to be considered for method calls, since it is ensured by the JAVA compiler that these are always caught within the method body. Only exceptions can be visible after the method call.

Qualified method calls

A typical language construct for object-oriented languages is the qualified method call `o.m()`, where method `m` in object `o` is called[3]. Before actually executing the method body, first the

[3]Notice that `o` can be `this`.

appropriate method has to be selected. Which method is selected depends on the run-time type of the object. Here we present a proof rule for this dynamic binding. Proof rules for late binding are not discussed in [Apt81], but they can be found in [PHM99, Ohe00].

In our semantics, qualified calls `o.m()` are translated by using special functions as CS2S (see Section 2.6.10). For example, if `o` is statically declared in class `A`, then `o.m()` translates to CS2S(A_clg)([[o]])([[m()]]). Following closely the evaluation strategy of these functions, appropriate proof rules can be formulated. For example, the following proof rule for partial correctness of CS2S is sound in our semantics.

– TYPE THEORY ———————————————————————————

$$\frac{\begin{array}{l}\{\lambda x : \mathsf{OM}.\ R\,x\ \wedge\ x = z\}\ \mathit{statement}(\mathit{coalg}(\mathit{name}\,z)(\mathit{refpos}\,z))\ \{Q\}\end{array}}{\{P\}\ \mathsf{CS2S}(\mathit{coalg})(\mathit{ref\_expr})(\mathit{statement})\ \{Q\}}$$

where

$$\begin{array}{l}\exists \mathit{refpos} : \mathsf{OM} \to \mathsf{MemLoc}.\ \exists \mathit{name} : \mathsf{OM} \to \mathsf{string}.\ \forall z : \mathsf{OM}.\\ \quad \{P\}\\ \quad \mathit{ref\_expr}\\ \quad \{\lambda x : \mathsf{OM}.\ \lambda v : \mathsf{RefType}.\ \ R\,x\ \wedge\\ \qquad\qquad\qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{false}\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{ref}\,r \mapsto\ \ r = \mathit{refpos}\,x\ \wedge\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{get\_type}\,r\,x = \mathit{name}\,x\}\}\end{array}$$

To avoid the problem that the logical variables cannot be instantiated if the state space is unknown, they are parametrised over the state space.

Once this rule has been applied, the actual late binding is done. In our semantics this is encoded by the coalgebra, parametrised by memory position and name. If evaluation of the reference expression produces a concrete name, the appropriate method can be looked up. Otherwise reasoning has to be done with the method specification.

Comparing this rule with the rules presented in [PHM99] reveals that this rule roughly corresponds to their invocation rule (where `T:m` denotes a method `m` which is subject to late binding, statically declared in (a superclass of) class `T` and `y` is a program variable with static type `T`).

$$\frac{\{P\}\,\mathtt{T:m}\,\{Q\}}{\{y \neq \mathtt{null}\ \wedge\ P[y/\mathtt{this}, e/p]\}\ \mathtt{x = y.T:m(e);}\ \{Q[x/\mathit{result}]\}}$$

An important difference between their and our approach is that they reason at a syntactic level, while we reason at a semantic level. In our semantics, the expression `x = y.T:m(e)` translates into A2E(x_becomes)(CE2E(T_clg)(y)(m(*e*))). Thus our rule is more general, because the method call can appear in any context, and the receiver object can be expressed by an arbitrary expression, but this difference is not essential: the rule by Poetzsch-Heffter and Müller can easily be adapted in this way. The rule states the following. Suppose that $\{P\}\,\mathtt{T:m}\,\{Q\}$ is established for the method `m`. This means that for all possible implementations of `m` in `T` or in subclasses of `T` $\{P\}\,\mathtt{m}\,\{Q\}$ holds. If `m` is called in a concrete object `y`, then it has to be shown that `y` is non-null and P is true for this object – thus in P `this` is replaced by the current object `y`, and the actual parameters are substituted. If this precondition can be shown, then Q is known to hold, and because of the assignment the *result* is replaced by `x`.

Poetzsch-Heffter and Müller also present rules (the class-rule and subtype-rule) to formally establish the correctness of the method $\{P\}\,\mathtt{T:m}\,\{Q\}$. Basically, they require that it is shown that the (run-time) type of `y` is a subtype of `T` and that for all possible subtypes of `T`, $\{P\}\,\mathtt{m}\,\{Q\}$ holds. If the class hierarchy is not open to extensions, then $\{P\}\,\mathtt{T:m}\,\{Q\}$ can be concluded from this.

Von Oheimb [Ohe00] also presents a proof rule for dynamic binding. This rule basically states the following (leaving out issues of argument evaluation, local variables *etc.*): to show $\{P\}\,\mathtt{o.m()}\,\{Q\}$ with `T` the static type of `o`, one has to show that for all classes `D` the following holds.

$$\{P \ \wedge \mathsf{SubClass?}\,\mathtt{D}\,\mathtt{T}\}\,\mathtt{m}_{impl_D}\,\{Q\}$$

Thus, for all implementations of `m` in subclasses of `T` $\{P\}\,\mathtt{m}\,\{Q\}$ has to be established. The user does not have to show that `o` is actually an instance of subclass of `T`. In Von Oheimb's approach this follows from JAVA type safety (see [ON99]).

Both approaches require that for every possible implementation of `m` it is shown that it satisfies the appropriate pre-post-condition relation (unless the precondition explicitly restricts which method implementations have to be considered). This implicitly requires that all possible implementations of `m` are known. If one reasons about an open program (as is done in this thesis) not all possible implementations of a method are known. In that case, one has to reason with the method specification of `m`. To verify a statement `o.m()` (with `o` static in `A`) the specification of `m` in `A` is used as an assumption. Independently, a verifier of class `A` or a subclass of class `A` has to show that `m` satisfies this specification. For more information on this approach, see Section 6.4.

5.6 Verification of an example program in PVS

To demonstrate the use of Hoare logic with abrupt termination, we consider the verification of a pattern match algorithm in JAVA. Chapter 7 discusses more verifications with Hoare logic (both in PVS and in ISABELLE). Consider the following algorithm, which is based on a pattern match algorithm described in [Par83].

– JAVA

```
class Pattern {
  int [] base;
  int [] pattern;
  int find_pos () {
    int p = 0, s = 0;
    while (true)
      if (p == pattern.length) return s;
      else if (s + p == base.length) return -1;
           else if (base[s + p] == pattern[p])
                  p++;
                else { s++; p = 0; }
  }
}
```

The `it-ti` construction proposed by Parnas [Par83] is programmed in JAVA as a `while` loop, with a condition which always evaluates to true. The loop is exited using one of two `return` statements. Explicit continues, as used in [Par83], are not necessary, because the loop body only consists of one `if` statement. In [Lei95, Chapter 5] a comparable algorithm is presented which searches the position of an element in a 2-dimensional array via two (nested) while loops. If the element is found, an exception is thrown, which is caught later. This has the same effect as a return. The algorithm is derived from a specification, using appropriate rules for exceptions.

This `find_pos` algorithm in itself is not particularly spectacular, but it is a typical example of a program with a while loop, in which a key property holds upon abrupt termination (caused by a `return`). The task of the algorithm is that, given two arrays `base` and `pattern`, it should determine whether `pattern` occurs in `base`, and if so, the starting position of the first occurrence of `pattern` should be returned. The algorithm checks – in a single while loop – for each position in the array `base` whether it is the starting point of the pattern – until the pattern is found. If the pattern is found, the while loop terminates abruptly, because of a return.

In the verification of this algorithm, it is assumed that both `pattern` and `base` are non-null references. In the proof our Hoare logic rules are applied as much as possible. The invariant, variant and exit condition are briefly discussed.

Some basic ingredients of the invariant for this while loop are:

- the value of the local variable `p` ranges between 0 and `pattern.length`;
- the value of `s` + `p` ranges between 0 and `base.length`, so that the local variable `s` is always between 0 and `base.length` − `p`;
- for every assumed value of `p`, the sub-pattern `pattern[0]`, ... , `pattern[p-1]` is a sub-array of `base`;
- for all i smaller than `s`, i is not a starting point for an occurrence of `pattern` (*i.e.* `pattern` has not been found yet).

To prove termination of the while loop, a variant with codomain nat × nat is used, namely (`base.length` − `s`, `pattern.length` − `p`). If the loop body terminates normally, the value of this expression strictly decreases, with respect to the lexical order on nat × nat. Either `s` is increased by one, so that the value of `base.length` − `s` decreases by one, or `s` remains unchanged and `p` is increased by one, in which case the value of the first component remains unchanged and the value of the second component decreases.

The exit condition states the following. If pattern occurs, then `p == pattern.length` and the value `s`, which is the starting point of the first occurrence of `pattern`, is returned. Otherwise, if the pattern does not occur, `s` = `base.length` and −1 is returned. Being able to handle such exit conditions is a crucial feature of the Hoare logic described in this chapter.

The correctness of this algorithm is shown in PVS in two lemmas. The first lemma states that if the `pattern` occurs in `base`, its starting position is returned, the other lemma states that if `pattern` does not occur, −1 is returned. Both proofs consists of approximately 250 proof commands. The crucial step in the proof is the application of the total return while rule with appropriate invariant. Rerunning the proofs takes approximately 5000 seconds on a Pentium II, 300 MHz.

5.7 Conclusions

We have presented the essentials of a Hoare logic for JAVA with side-effects and abrupt termination. In particular, it features rules for total correctness of abruptly terminating loops. Being able to reason about abrupt termination is crucial for verification of JAVA programs. This logic allows one to prove under which conditions exceptions will be thrown. This is essential information to use classes correctly as components.

The Hoare logic presented here is sound *w.r.t.* our JAVA semantics. It has been used in several example verifications (see Chapter 7). Using the proof rules in actual verification helped in developing and fine-tuning them, so that they are suited for use in a theorem prover.

The rules that have been presented here are only a small subset of all the rules that can be proven for JAVA. Appendix A presents a more complete overview of the rules for normal correctness (of statements and expressions), exception correctness (of statements and expressions), and return correctness. The rules for break correctness and continue correctness are similar to the rules for return correctness. The construction of these rules is straightforward, building on the ideas presented in this chapter.

Currently, an adaptation of this Hoare logic is under development, where the postcondition is replaced by a labeled product, containing postconditions for all termatination modes [JP00a]. The adapted proof rules and their soundness proofs build on the logic presented in this chapter.

Chapter 6

Class specification and the Java Modeling Language

Before a class can be verified, it first has to be clear what exactly requires verification: the desired properties have to be specified. This chapter introduces a language JML, short for JAVA MODELING LANGUAGE [LBR98], which can be used to write such class specifications for JAVA. From a clients perspective the specifications describe properties that can be assumed, but from the providers perspective they represent (proof) obligations, because the provided code is supposed to satisfy these properties. This means that to verify a method, one has to show that it satisfies its specification. In this verification, it can be assumed that the methods that are invoked from the "method under verification" are correct, *i.e.* these methods satisfy their specification. The correctness of a method can thus be established locally, assuming everything else behaves as specified. This is called modular verification, because the verification of a complete system can be split up into the verification of different components or modules.

JML is a so-called behavioural interface specification language, following the tradition of EIFFEL and the well-established design by contract approach [Mey97]. A programmer can annotate JAVA code with specifications in JML, using the special annotation markers `//@` and `/*@ ... @*/`. For a JAVA compiler these annotations are ordinary comments, so the annotated JAVA code remains valid. The annotations use the syntax for JAVA expressions, so that they are easy to read and write for JAVA programmers. In this chapter we will only mention a subset of all specification declarations availabe in JML. For more information, see [LBR98].

The LOOP compiler is currently being extended, so that appropriate proof obligations can be generated for an annotated JAVA program. These proof obligations are formulated in terms of the Hoare logic, presented in Chapter 5. To generate appropriate proof obligations, a formal semantics of the annotations has to be established. This is on-going research [BPJ00]. The Hoare logic described in Chapter 5 forms the basis for this semantics. In the case studies described in Chapter 7, JML annotations are used to express properties about the verified JAVA programs. Within these case studies, the translation from JML annotations to Hoare logic sentences is done by hand, but in the future this will be done by the LOOP compiler. The modular verification techniques that are described in this chapter form the basis for the verifications in the next chapter.

This chapter is organised as follows. Section 6.1 introduces the basic specification declarations of JML: behaviour specifications and class invariants. Section 6.2 discusses which proof obligations are generated from the behaviour specifications and invariants. Section 6.3

introduces model variables, which can be used to provide some means of data abstraction. Section 6.4 discusses how (JML) specifications can be used for modular verification. Section 6.5 discusses another specification declaration, so-called modifies clauses, which can be used to specify the side-effects of a method. Finally, Section 6.6 presents conclusions.

6.1 The Java Modeling Language (JML)

6.1.1 Predicates in JML

The predicates used in JML are built from ordinary JAVA expressions extended with logical operators, such as equivalence, `<==>`, and implication, `==>`, and with the existential and universal quantifiers, `\exists` and `\forall`, respectively. Also some new expression syntax is added: in the post-condition `\old(`E`)` denotes the value of the expression E in the "pre-state" of a method (*i.e.* in the state before method execution is started), `\result` denotes the result of a non-`void` method, and `\throws` denotes an exception, possibly thrown by the method.

Predicates in JML are required to be side-effect free, and therefore they are not allowed to contain assignments, including the increment and decrement operators, `++` and `--`. Methods may be invoked in predicates only if they are pure, *i.e.* terminate normally, and do not modify the state.

Requiring that predicates are side-effect free does not imply that predicates always terminate normally. Consider the predicate `a.length >= 0`, for `a` an array. If this predicate is evaluated in a state where `a` is a null reference, it will terminate abruptly with a `NullPointerException`. To prevent this kind of abrupt termination, an extra conjunct has to be added to the predicate: `a != null && a.length >= 0`.

6.1.2 Behaviour specifications

In JML behaviour specifications can be written for methods and constructors. We concentrate on methods. In JML three kinds of behaviour specifications are supported, namely `normal_behavior`, `exceptional_behavior` and `behavior` specifications. If a method has a `normal_behavior` specification, then it should terminate normally, assuming the precondition holds. Similarly, an `exceptional_behavior` prescribes that a method must terminate abnormally, and a `behavior` specification that the method sometimes terminates normally and sometimes abnormally.

For example, consider the following `normal_behavior` specification for a method `m`.

– JML

```
void m();
/*@ normal_behavior
  @   requires: P;  // P is a predicate
  @   ensures : Q;  // Q is a relation, relating
  @                 // the method's pre-state and
  @                 // post-state.
  @*/
```

The basic ingredients of a `normal_behavior` are its pre-condition, in JML called the `requires` clause, and its post-condition, the `ensures` clause. This `normal_behavior` specification is a *total correctness assertion*: it says that if `P` holds in a state x, then method `m` executed in state x will terminate normally, resulting in state y where `Q`(x, y) holds. The pre-state x is needed in the post-condition because `Q` may involve an `\old(-)` expression for evaluation in the pre-state.

A `behavior` specification can consist of the two abovementioned clauses, extended with a `signals` clause:

– JML

```
void m();
/*@ behavior
  @   requires: P;
  @   ensures : Q;
  @   signals : (E) R;
  @*/
```

The `signals` clause is the post-condition in case of abrupt termination of method `m`. This example specification is a conjunction of two *partial correctness Hoare sentences*. The first one says that if `P` holds in a state x and method `m` executed in state x terminates normally resulting in a state y, then `Q`(x, y) should hold. The second one says that if `P` holds in a state x and method `m` executed in state x terminates abruptly with an exception of type `E'` in a state y, then `R`(x, y) holds and `E'` should be a subclass of `E`.

Similarly, an exceptional behaviour contains a `requires` and a `signals` clause.

– JML

```
void m();
/*@ exceptional_behavior
  @   requires: P;
  @   signals : (E) R;
  @*/
```

It is interpreted as a total exception correctness Hoare sentence, thus if the method is executed in a state x satisfying the precondition `P`, it terminates abruptly, because of an exception `E'` in a state y, where `R`(x, y) holds and `E'` is a subclass of `E`.

A method annotation can consist of several behaviour specifications, combined with the keyword `also`. As an example of an annotated method, we look at the method `firstElement`, returning the first element in an array `arg` of `Objects`.

– JML

```
/*@ exceptional_behavior
  @   requires : arg == null;
  @   signals  : (NullPointerException) true;
  @ also
  @ behavior
  @   requires : arg != null;
  @   ensures  : \result = arg[0];
```

```
  @   signals  : (ArrayIndexOutOfBoundsException)
  @                arg.length == 0;
  @*/
Object firstElement (Object [] arg) {
  return arg[0];
}
```

This specification says that if the argument array `arg` is `null` a `NullPointerException` will be thrown, otherwise there are two possibilities: the value of `arg[0]` is returned or an `ArrayIndexOutOfBoundsException` is thrown, in which case it can be proven that `arg.length` is 0.

6.1.3 Invariants

Recall from Section 2.6.3 that an invariant is a predicate on states which always holds, as far as an outsider can see: an invariant holds immediately after an object is created and before and after a method is executed, but during a method's execution it need not hold. Invariants restrict the possible values of the fields of an object (in the visible states). To prove that a certain predicate is an invariant, one proves that (1) the predicate holds after object creation, and (2) it is preserved by every method, *i.e.* the predicate holds after (normal or abnormal) termination of a method, assuming that it holds when the method's execution starts.

An example of a (trivial) JML invariant is:

– JML
```
class A {
  //@ invariant: true;
  ...
}
```

JML offers the possibility to write multiple invariants within one class. They can be transformed into a single invariant via conjunctions.

6.2 Proof obligations

As already mentioned, invariants and behaviour specifications give rise to proof obligations. They can be expressed in our extended Hoare logic, as described in Chapter 5, although some minor changes are required.

In the generation of proof obligations from the method annotations, the pre- and postconditions and the invariants are translated as JAVA expressions into state transformer functions from OM to ExprResult[OM, bool]. These translated expressions are composed with appropriate functions which map the results of evaluating the expression to Boolean values, so that their compositions are predicates on the state space. Here we abstract away from this mapping function, for more information see [BPJ00].

The `ensures` clauses of non-`void` methods can contain a special variable result, denoting the return value of the method. Remember that post-conditions of Hoare logic sentences over expressions are predicates over the state space and the type of the return value. Thus, every occurrence of result is replaced by this return value.

The same approach is taken for `signals` clauses, which can contain a special `\throws` keyword, representing the exception that occurred in the method. These `signals` clauses are translated as predicates over states and exceptions (elements in RefType).

The last special syntactic construct of JML that has to be incorporated into our Hoare logic is the `\old(-)` expression, which refers to the pre-state. For this we use so-called logical variables (like z below) and we allow post-conditions to be relations over the pre- and the post-state. Assuming that z is a logical variable of type OM, representing the pre-state, the following translation is used.

$$[\![\texttt{\textbackslash old}(E)]\!] \stackrel{\text{def}}{=} [\![E]\!](z)$$

For example, the normal behaviour specification for `m` above (page 143), together with an invariant I, yields the following proof obligation for `m`[1].

– TYPE THEORY –

$$\forall z : \mathsf{OM}.\ [\lambda x : \mathsf{OM}.\ I\ x \wedge P\ x \wedge z = x]\ \ m\ \ [\lambda y : \mathsf{OM}.\ I\ y \wedge Q(z, y)].$$

Similarly, the behaviour specification yields a conjunction of two partial Hoare sentences:

– TYPE THEORY –

$$\begin{array}{l}\forall z : \mathsf{OM}. \\ \quad \{\lambda x : \mathsf{OM}.\ I\ x \wedge P\ x \wedge z = x\} \\ \quad\ \ m \\ \quad \{\lambda y : \mathsf{OM}.\ I\ y \wedge Q(z, y)\} \\ \quad\ \ \bigwedge \\ \quad \{\lambda x : \mathsf{OM}.\ I\ x \wedge P\ x \wedge z = x\} \\ \quad\ \ m \\ \quad \{\mathsf{exception}(\lambda y : \mathsf{OM}.\ \lambda E' : \mathsf{RefType}.\ I\ y \wedge R(z, y)(E'), E)\}\end{array}$$

Finally, the exceptional behaviour specification yields a single Hoare sentence.

– TYPE THEORY –

$$\begin{array}{l}\forall z : \mathsf{OM}.\ \ [\lambda x : \mathsf{OM}.\ I\ x \wedge P\ x \wedge z = x] \\ \qquad\qquad\ \ m \\ \qquad\qquad [\mathsf{exception}(\lambda y : \mathsf{OM}.\ \lambda E' : \mathsf{RefType}.\ I\ y \wedge R(z, y)(E'), E)]\end{array}$$

As an example, we look at the proof obligations that are generated for the method `firstElement` (forgetting about possible class invariants).

[1] In general it is not sufficient to assume that only the invariant of the current class holds; one also needs that the invariants of all the objects that can be referenced holds [PH97].

$$\forall z : \mathsf{OM}.\ \forall \mathsf{arg} : \mathsf{RefType}.$$
$$[\lambda x : \mathsf{OM}.\ \mathsf{arg}\, x == \mathsf{null} \wedge z = x]$$
$$\mathsf{firstElement}(\mathsf{arg})$$
$$[\mathsf{exception}(\mathsf{true}, \text{“NullPointerException”})]$$

$$\forall z : \mathsf{OM}.\ \forall \mathsf{arg} : \mathsf{RefType}.$$
$$\{\lambda x : \mathsf{OM}.\ \mathsf{not}(\mathsf{arg}\, x == \mathsf{null}) \wedge z = x\}$$
$$\mathsf{firstElement}(\mathsf{arg})$$
$$\{\lambda x : \mathsf{OM}.\ \lambda v : \mathsf{RefType}.\ v == \mathsf{access\_at}(\mathsf{get\_ref})(\mathsf{arg}, 0)\, x\}$$
$$\wedge$$
$$\{\lambda x : \mathsf{OM}.\ \mathsf{not}(\mathsf{arg}\, x == \mathsf{null}) \wedge z = x\}$$
$$\mathsf{firstElement}(\mathsf{arg})$$
$$\{\mathsf{exception}\ (\lambda x : \mathsf{OM}.\ \lambda E : \mathsf{RefType}.\ \mathsf{arg.len} = 0,$$
$$\texttt{"ArrayIndexOutOfBoundsExceptions"})$$

The proof rules for the extended Hoare logic can be used to prove these JML obligations. The case studies in the next chapter give some more examples.

6.3 Model variables

An important question is how to write specifications for a method so that they give enough information to be useful in the verification of other methods, without relying on too many implementation details. Often, methods have an effect on the internal state space of an object, which is hidden from clients of a class, but which is important to describe their behaviour. It even can be the case that the static type of the receiver object of a method call is an interface or abstract class, which does not contain (all of) the fields. Therefore, so-called model variables or abstract variables, which represent a set of concrete variables, are used to write the specifications. These model variables can be publicly visible. To verify a concrete class, *i.e.* a class of which instances can be created, a representation function has to be given which maps the values of the fields to the values of the model variables. The use of model variables is an extension of Hoare's data abstraction technique [Hoa72].

In JML model variables are preceded by a special keyword `model`. If a model variable is declared in a class `C`, it does not actually occur in the implementation of the class, but for purposes of specification every instance of `C` is imagined to have such a field. Model variables can have primitive types or reference types. If a model variable has a reference type, this should always be a so-called pure class, *i.e.* a class in which the methods do not have side-effects. In that case the methods of these class can safely be used in the specifications. There is a collection of pure classes available which can be used as types for the model variables.

As an example we consider part of the specification of an unbounded stack from [LBR98].

– JML

```
public abstract class UnboundedStack {

  /*@ public model JMLObjectSequence theStack
```

```
    @*/

  //@ public invariant: theStack != null;

  ...

  /*@ public_normal_behavior
    @   requires: !theStack.isEmpty();
    @   ensures: \result == theStack.first();
    @*/
  public abstract Object top( );
}
```

This specification starts by declaring a model variable `theStack` which is in the class `JMLObjectSequence`, *i.e.* a sequence of objects. The model variable is used in the specification of the class invariant and the methods. Methods from the class `JMLObjectSequence` can be used in the specifications. The class `JMLObjectSequence` is thus used to give a model of the class `UnboundedStack`.

Suppose that we construct a class which is a concrete implementation of the `UnboundedStack` specification. To verify our implementation, *i.e.* to show that it satisfies its specification, the fields of the implementation have to be related to the model variables. This is done by so-called represents clauses. For example, our implementation could contain the following lines, stating that the value of the field `size` is equal to the length of `theStack`.

– JML

```
int size;
//@ public represents: size <- theStack.length();
```

Sometimes it is not possible to give an exact representation function, therefore dependency clauses are introduced [Lei95]. If a model variable `a` depends on a variable `b` (either concrete or abstract), this means that every time the value of `b` changes, the value of `a` may have changed.

When proving the correctness of implementations (within a theorem prover), the methods that are called on the model variable `theStack` (in the specifications) will have to be evaluated. It is still an open question how this is done best:

- by using the (translated)[2] specifications of the methods in `JMLObjectSequence`,
- by using a (LOOP translated) JAVA implementation of the methods in `JMLObjectSequence`, or
- by reasoning in the logic of the theorem prover immediately, thus mapping the method calls to operations in the logic instead of to their JAVA implementations.

In the verification of class `AbstractCollection` (Section 7.2 we choose the last option[3].

[2] into the logic of the theorem prover

[3] Actually, we go even further by leaving out the intermediate step of the pure class, since our model variables have ISABELLE types. This is possible because we do the translation from JML specification to ISABELLE by hand. Despite this simplification, we still get all the typical problems involved with modular verification.

6.4 Modular verification

It is typical for the verification of large programs that one would like to verify smaller parts in isolation, without knowing anything about the implementation of the other parts. Instead of taking the whole system into account, only a small part of the implementation should be relevant for the verification. This is usually called modular verification. The challenge in modular verification is to do this in such a way that from the correctness of the components (the modules), the correctness of the whole system can be concluded. Research has been focusing on sound methods of modular verification. It is impossible to find a complete method for modular verification [Lei95].

For verification of object-oriented programs, modular verification is even more essential. Often one wishes to verify a single class that can be used in different contexts, where the surrounding classes have different implementations. Actually, when verifying a particular method, one should not even rely on the implementation of the other methods in the same class, because in subclasses they might be overridden.

This is typically the case with (multi-purpose) classes from an object-oriented library, which can be plugged into arbitrary programs. Instead of reverifying them within each application (which is the responsibility of the application developer), they should be verified in isolation (by the library developer). The application designer can then rely on the correctness of the library class, when building (and possibly verifying) the application.

This section discusses how modular verification can be used in the LOOP project. Several papers have appeared discussing aspects of modular verification for object-orientation and JAVA. This discussion is based on these papers (in chronological order) [Lea93, LW94, Lei95, DL96, LS97, MPH97, DLN98, Lei98, PHM98, LBR99, LD00, MPH00b].

6.4.1 Reasoning with specifications

Suppose that one wishes to verify a method `m` that calls another method `n` (on some object `o`, which may be `this`). At verification time, only the static type of the object `o` is known, thus it cannot be determined what the implementation is of the method that actually will be called (since this is subject to late binding).

A typical example where this late binding problem occurs is the container classes, which are used to represent a collection of objects. In advance, the only thing that is known about these objects, is that they are subclasses of class `Object`, and thus that they provide an implementation for `equals` (as `Object` provides an implementation for `equals`). Typically, this method is overridden in subclasses, to deal with structural equivalence of objects. To test membership of an object in a container, this `equals` method will be used. To verify the correctness of such a container membership operation, abstract properties describing the `equals` operation have to be used. This is what is done for example in the verifications of the methods `remove` from `AbstractCollection` and `toString` and `indexOf` from `Vector`, see Chapter 7.

To verify methods which call other methods, this method call has to be taken into account. It cannot be ignored. Even though the implementation is unknown, a specification of the method can be given. This method specification *i.e.* its pre-post-condition behaviour and possible class invariants, can be used in the verification of other methods, calling this method. For example, when verifying method `m`, which contains a call to a method `o.n()`, with `o` declared as an

instance of class A, the specification of n in the static type A is used. The verifier of m first has to show that the precondition of n is satisfied, and then can use the postcondition of n in the remainder of the verification.

6.4.2 Behavioural subtypes

Of course, using specifications to reason about method calls only makes sense if the actual implementations of the method that can be called at run-time satisfy this specification. If a method contains a call to o.n() where o is declared in class A, then at run-time o is always in class A or in a subclass of A. Thus, to ensure that all possible implementations ensure the specification, it has to be shown that in all subclasses of A, the implementation of method n satisfies the specification of n in A. If this is the case, then the verification of m, using the specification of o.n() remains valid (and the behaviour of m remains as expected).

In more general terms: it should be shown that wherever a superclass is declared, an instance of a subclass might be used and this will not present any unpredicted behaviour. All the methods in a subclass should preserve the behaviour of the methods in a superclass. If this is the case, an instance of a subclass cannot be distinguished from an instance of the superclass, as long as only methods from the superclass are used.

To express this, the notion of behavioural subtype is introduced [Mey97, Ame90, LW94, Pol00]. Classes can only be behavioural subtypes, if their signatures are subtypes. Furthermore, methods in the subtype that are overriding (or redefining) a method of the supertype, should preserve the behaviour of the method of the supertype. In JAVA a subclass overrides a method from a superclass if it contains an implementation for a method with the same name and exactly the same signature[4]. The JAVA compiler also accepts methods with the same name, but different argument types, but this only leads to overloading of method names. Overloaded methods are considered as different methods by the JAVA compiler, and it is statically decidable which method is actually intended.

Behavioural subtype: Suppose we have two classes C and D. Class D is a behavioural subtype of class C if the following conditions hold.

1. The class invariant of class D implies the class invariant of class C

$$\forall x : \mathsf{OM}.\ \mathsf{invariant}_D\ x \supset \mathsf{invariant}_C\ x$$

2. Subtype methods preserve the behaviour of supertype methods, *i.e* for all methods m_C that are overridden by m_D, the following holds.

   - $\forall x : \mathsf{OM}.\ \mathsf{pre}_{m_C}\ x \supset \mathsf{pre}_{m_D}\ x$
   - $\forall x : \mathsf{OM}.\ \mathsf{post}_{m_D}\ x \supset \mathsf{post}_{m_C}\ x$

Notice that this notion of behavioural subtyping gives proof obligations for each (overriding) method to show that it is a behavioural subtype of the method in the superclass. As pointed out by Dhara and Leavens [DL96], one can also interpret the annotations of a subclass in such a way that it is a behavioural subtype by construction. For example, one can interpret the postcondition of method m in subclass D as the conjunction of the postcondition of method m in superclass

[4]The overriding method may declare less exceptions throwable than the method in the superclass.

C and the postcondition-annotation of m in D. It is then trivial to show that the (interpretation of the) postcondition of m in D implies the postcondition of m in C. This is called **inheritance of specification**. This is similar to the interpretation of method annotations in Eiffel [Mey97].

As explained above, a typical example of a method for which the behavioural subtype approach is used is `equals` from `Object`. In `Object` this method is implemented by testing for reference equality only. In subclasses this method is often overridden to deal with structural equivalence of objects. The JML specification of `equals` thus has to take this possibility of overriding into account.

– JML

```
/*@ normal_behavior
  @   requires: true;
  @    ensures: this == obj ==> \result &&
  @             obj == null ==> !\result;
  @*/
public boolean equals(Object obj)
```

If the argument is the same reference as the receiving method, the result of the method should be true. If the argument is a null reference, the result should be false (because the receiving object cannot be null). Otherwise, the outcome is not specified. The implementation of `equals` in `Object` satisfies this specification. Subclasses that override this method can define their own notion of (structural) equivalence, as long as their implementation still satisfies this specification of `equals`. Furthermore, we also specifiy that the `equals` operation is symmetric and transitive (on non-null references).

6.4.3 Representation exposure

A typical problem that has to be dealt with in modular verification is the problem of representation exposure or pointer leaking. If there are more references to one object, changes to this object via one reference may affect the correctness of the objects holding other references.

Consider for example the following class `Rectangle`, with methods `minX()`, `maxX()`, `minY()` and `maxY()`, returning the minimal and maximal x and y-coordinates of the rectangle, respectively[5]. Now suppose that we have another class, which draws something in the rectangle.

– JAVA

```
class Draw {
  Rectangle r;
  int x, y;

  ..
}
```

A typical invariant for this class (in JML notation) would be the following, stating that the values of `x` and `y` are always between the borders of the rectangle.

[5]This example is due to Leino and Stata [LS97].

– JML
```
/*@ invariant: r != null &&
  @            r.minX() <= x & x <= r.maxX() &
  @            r.minY() <= y & y <= r.maxY()
  @*/
```

As explained above, in the verification of class `Draw` the pre- and postconditions of the methods in `Rectangle` are used. Possible subclasses of `Rectangle` do not break the correctness of `Draw`, as long as they are behavioural subtypes.

Unfortunately, correctness of the class `Draw` is still not completely secured. Suppose that their exists another reference to the `Rectangle` field `r` in `Draw`. If this reference is not visible from within `Draw`, this can easily break the correctness. Via this other reference, the state of `r` might be changed in such a way that the invariant of `Draw` becomes invalid. To avoid this problem, it should be guaranteed that `r` cannot 'leak' out of the scope of `Draw`. The transfer of modifiable components across abstraction boundaries (in our case: class boundaries) is called representation exposure [DLN98] (or rep exposure for short). Several solutions have been proposed to deal with rep exposure [DLN98, MPH00b], but there is no complete and easy solution yet.

Most JAVA library classes have been constructed in such a way that they do not leak pointers. If references are returned by methods, they are usually fresh pointers (obtained via cloning, for example). Therefore, in the case studies in Chapter 7 the problem of representation exposure is not relevant.

6.5 Changing the state: the frame problem

Unfortunately, using only the functional specification of a method usually is not enough to reason about arbitrary method calls. Suppose that we verify the following (silly) class.

– JAVA
```
class C {
  int [] a;

  /*@ normal_behavior
    @   ensures: a.length >= 4;
    @*/
  void m () {
    a = new int [5];
    n ();
  }

  /*@ normal_behavior
    @   ensures: true;
    @*/
  void n () {
  }
}
```

The method `n` may be overridden in subclasses of `C`, thus in the verification of method `m` the specification of `n` is used. However, to establish the postcondition of `m` we need to know that `n` does not change the length of the array `a`. Using only its functional behaviour is not enough to establish this. Therefore, so-called modifies clauses are introduced, using the keyword `modifiable:` in JML. A modifies clause in a method specification states which variables may be changed by a method; all other variables must remain unchanged.

A modifies clause may contain a model variable. In that case, it means that all variables on which this abstract variable depends may change. In contrast, if a modifies clause mentions a concrete field, but not an abstract variable depending on this field, then this field may change only in such a way that it does not affect the value of the abstract variable.

Modifies clauses should also be taken into account when deciding whether a class is a behavioural subtype. It is not immediately clear what the corresponding proof obligations for a modifies clause should be. Suppose that extra fields are defined in the subclass. Should overriding methods be allowed to modify these new fields? This question is often referred to as the frame problem. Often modifies clauses are translated into extra postconditions, stating which values should remain the same. In behavioural subtypes postconditions in subclasses should be stronger than those in superclasses. But then, the postcondition would only allow fewer variables to change, not more, and this is not what we want. Of course, we could also say that all newly declared fields might be changed, but this is often too liberal and might prevent verification of some class which explicitly uses the subclass. Several solutions have been proposed, using extra annotations to group variables [Lei98] or by restricting dependencies between the variables [MPH00b]. For the verifications in the case studies in Chapter 7 this problem is not relevant, because no new fields are declared in subclasses.

6.5.1 Side-effect freeness

Another question related to `modifiable:` clauses is what it actually means for methods not to have side-effects. We take the following view: a method does not have side-effects if it does not change the already allocated memory. A side-effect free method may thus allocate new memory on the heap. We define special abbreviations which define when the heap, stack and static memory are considered equal, respectively.

– TYPE THEORY

$$x, y : \mathsf{OM} \vdash$$
$$\mathsf{heap\_equality}(x, y) : \mathsf{bool} \stackrel{\text{def}}{=}$$
$$\mathsf{heaptop}\, x \leq \mathsf{heaptop}\, y \wedge$$
$$\forall t : \mathsf{MemLoc}.\, t < \mathsf{heaptop}\, x \supset \mathsf{heapmem}\, x\, t = \mathsf{heapmem}\, y\, t$$

$$x, y : \mathsf{OM} \vdash$$
$$\mathsf{stack\_equality}(x, y) : \mathsf{bool} \stackrel{\text{def}}{=}$$
$$\mathsf{stacktop}\, x = \mathsf{stacktop}\, y \wedge$$
$$\forall t : \mathsf{MemLoc}.\, t < \mathsf{stacktop}\, x \supset \mathsf{stackmem}\, x\, t = \mathsf{stackmem}\, y\, t$$

$$x, y : \mathsf{OM} \vdash$$
$$\mathsf{static\_equality}(x, y) : \mathsf{bool} \stackrel{\text{def}}{=}$$
$$\forall t : \mathsf{MemLoc}.\ \mathsf{staticmem}\ x\ t = \mathsf{staticmem}\ y\ t$$

Two states are called equal if heap_equality, stack_equality and static_equality hold for them. Notice that heap_equality is not influenced by newly created objects, which are stored above the old heaptop. A method is called side-effect-free if its pre- and post-state are always equal in this sense.

– TYPE THEORY ———————————————————————————

$$m : \mathsf{OM} \to \mathsf{ExprResult[OM, Out]} \vdash$$
$$\mathsf{side\_effect\_free}(m) : \mathsf{bool} \stackrel{\text{def}}{=}$$
$$\begin{array}{l}
\forall x : \mathsf{OM}.\ \ \mathsf{CASE}\ m\ x\ \mathsf{OF}\ \{ \\
\qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{true} \\
\qquad\qquad \mid \mathsf{norm}\ y \mapsto \ \mathsf{heap_equality}(x, y.\mathsf{ns}) \wedge \\
\qquad\qquad\qquad\qquad\quad \mathsf{stack_equality}(x, y.\mathsf{ns}) \wedge \\
\qquad\qquad\qquad\qquad\quad \mathsf{static_equality}(x, y.\mathsf{ns}) \\
\qquad\qquad \mid \mathsf{abnorm}\ a \mapsto \ \mathsf{heap_equality}(x, y.\mathsf{es}) \wedge \\
\qquad\qquad\qquad\qquad\quad \mathsf{stack_equality}(x, y.\mathsf{es}) \wedge \\
\qquad\qquad\qquad\qquad\quad \mathsf{static_equality}(x, y.\mathsf{es})\ \}
\end{array}$$

A similar definition exists for void-methods.

6.6 Conclusions

This chapter sketches an annotation language for JAVA, called JML. JML allows to write specifications for JAVA classes. An implementation of a JAVA class is said to be correct if it satisfies its specifications. When verifying a class (or method), the specifications of the component classes can be used as assumptions in the correctness proof. This chapter also discusses several topics related to this modular kind of reasoning, such as behavioural subtyping, representation exposure and the frame problem.

Assertions in the annotation language JML are written in (extended) JAVA syntax, so that they are easy to read and write for JAVA programmers. Several annotation constructs have been discussed: method behaviour specifications (describing partial and total (exception) correctness of methods), class invariants, model variables, representation and dependency relations and modifiable clauses. Appropriate proof obligations for the methods can be generated on the basis of the method annotations, making use of our special Hoare sentences, tailored to JAVA.

As mentioned above, JML is used to write the specifications for the classes that are verified in the case studies described in Chapter 7. JML is also used for a follow-up specification and verification project focusing on the entire JAVA Card API [PBJ00] (which is much smaller than the standard JAVA API). In these projects, the JML specifications are added *post hoc*, after the JAVA code has already been written. It would have been much more efficient (for us, as verifiers) if the JML specifications would have been written together with (or even before)

the JAVA implementation. One of the main points behind JML (and this work) is that writing such specifications at an early stage really pays off. It makes many of the implicit assumptions underlying the implementation explicit (*e.g.* in the form of invariants), and thus facilitates the use of the code and increases the reliability of software that is based on it. Furthermore, the formal specifications are amenable to tool support, for verification purposes. It is our hope that certainly for crucial classes in standard libraries the use of specification in languages like JML (and subsequent verification) becomes standard. For such library classes, the additional effort may be justifiable.

Chapter 7

Two case studies: verifications of Java library classes

One of the reasons for the popularity of object-oriented programming is the possibility it offers for reuse of code. Usually, the distribution of an object-oriented programming language comes together with a collection of ready-to-use classes, in a class library or API (Application Programmer's Interface). Typically, these classes contain general purpose code, which can be used as a basis for many applications. Before using such classes, a programmer usually wants to know how they behave and when their methods terminate normally or throw exceptions. One way to do this, is to study the actual code. This is time-consuming and requires an understanding of all particular ins and outs of the implementation – which may even be absent, for native methods. Hence this is often not the most efficient way. Another approach is to study the (informal) documentation provided. As long as this documentation is clear and concise, this works well, but otherwise a programmer is still forced to look at the actual code.

One way to improve this situation is to formally specify suitable properties of standard classes, and add these specifications as annotations to the documentation. Examples of properties that can be specified are termination conditions (in which cases will a method terminate normally, in which cases will it throw an exception), pre-post-condition relations and class invariants. Chapter 6 describes a specification language tailored to JAVA, which allows one to write such annotations. Once sufficiently many properties have been specified, one only has to understand these properties, and there is no longer any need to study the actual code.

Programmers must of course be able to rely on such specifications. This introduces the obligation to actually verify that the implementation satisfies the specified properties. Even stronger, specifications can exist independently of implementations, as so-called interface specifications. As such they may describe library classes in a component-oriented approach, based on interface specifications regulating the interaction between components. In such a "design by contract" scenario the provider of a class implementation has the obligation to show that the specification is met. And naturally, every next version of the implementation should still satisfy the specification, ensuring proper upgrading. Thus, verification of class specifications is an important issue.

This chapter discusses two case studies, each involving a class from the standard JAVA class library. The first case study verifies a class invariant over the class `Vector`. This verification is done in PVS. The second case study uses ISABELLE to prove behavioural specifications for the methods in the class `AbstractCollection`, using specifications for the abstract

methods. In both case studies the actual verification takes the object-oriented character of JAVA into account: (non-final) methods may always be overridden, so that one cannot rely on a particular implementation. Instead, one has to reason from method specifications in such cases (see Section 6.4 for more information).

The `Vector` case study is presented in Section 7.1 and Section 7.2 presents the verification of the class `AbstractCollection`.

7.1 Verification of Java's Vector Class in PVS

This case study presents a verification of an invariant property for the `Vector` class from JAVA's standard library (API). The property says (essentially) that the actual size of a vector is less than or equal to its capacity. It is shown that this "safety" or "data integrity" property is maintained by all methods of the `Vector` class, and that it holds for all objects created by the constructors of the `Vector` class.

The `Vector` class is one of the library classes in the standard JAVA distribution [AG97, GJSB00, CLK98]. Object in the `Vector` class basically consist of an array of objects. According to needs, at run-time this array may be replaced by an array of different size[1] (but containing the same elements). The essence of the `Vector` invariant that is proven is that the size of a vector never exceeds the length of this internal array. Clearly, this is a crucial safety property.

The choice for the `Vector` class in this verification is in fact rather arbitrary: it serves our purposes well because it involves a non-trivial amount of code (including the code from its surrounding classes from the library), and gives rise to an interesting invariant. However, other classes than `Vector` could have been verified. And in fact, there are many classes in the JAVA API, like `StringBuffer` using an array of characters with a count, for which a similar invariant can be formulated. Thus the property that we consider is fairly typical as a class invariant.

The specification of the `Vector` invariant (and pre- and post-conditions for the methods of this class) are written in JML (introduced in Chapter 6). As explained, the LOOP tool is currently being extended to translate also JML specifications, which will give rise to specific proof obligations in Hoare logic. The JML specifications used in this case study have been translated by hand, into corresponding Hoare sentences (in PVS), which are used in verifications. For the verification, extensive use has been made of the Hoare logic, presented in Chapter 5.

This is one of the largest case studies done so far within the LOOP project. It demonstrates the feasibility of the formal approach to software development, as advocated in this project.

The case study is structured as follows. First the `Vector` class and its translation are discussed. Then the class invariant is discussed, and finally the verification of several methods is discussed in more detail.

7.1.1 Vector in Java

JAVA's `Vector` class[2] is part of the `java.util` package. It can be found in the sources of the JDK distribution. The class as a whole is too big to describe here in detail: it contains

[1] Arrays in JAVA have a fixed size; vectors are thus useful if it is not known in advance how many storage positions are needed.

[2] We use version number 1.38, written by Lee Boynton and Jonathan Payne, under Sun Microsystems copyright.

three fields, three constructors, and twenty-five methods. Most of the method bodies consist of between five and ten lines of code. We describe the interface of the `Vector` class, and also its "surrounding" classes in the JAVA library. The latter are classes used in the `Vector` class.

Interface of the Vector class

The `Vector` class has three fields, namely an array `elementData` with elementtype `Object` in which the elements of the vector are stored, an integer `elementCount` which holds the number of elements stored in the vector, and an integer `capacityIncrement` which indicates the amount by which the vector is incremented when its size (`elementCount`) becomes greater than its capacity (length of `elementData`). If `capacityIncrement` is greater than zero, every time the vector needs to grow the capacity of the vector is incremented by this amount, otherwise the capacity is doubled. These fields are all protected, so that they can only be accessed in (a subclass of) `Vector`.

The `Vector` class has three constructors, which all are public and thus can be accessed in any class. The constructor `Vector()` creates an instance of the `Vector` class by allocating the array `elementData` with an initial capacity of ten elements, and a capacity increment of zero. The second constructor `Vector(int initialCapacity)` takes an integer argument, which is the initial capacity, and sets the capacity increment to zero. The third constructor `Vector(int initialCapacity, int capacityIncrement)` takes two integer arguments, one for the initial capacity and the other for the capacity increment. After creating an instance of the `Vector` class the field `elementCount` is implicitly set to zero.

We do not describe all methods of the `Vector` class in detail. For that, the reader is referred to the standard documentation [CLK98] for more information, and only the interface of the `Vector` class is listed here, see Figure 7.1. The names and types give some idea of what these methods are supposed to do.

Surrounding classes

The `Vector` class implicitly extends the `Object` class. All fields and methods declared in the `Object` class are thus inherited. Of particular importance in the `Vector` class are the methods `equals`, `clone`, and `toString` from `Object`. These may be overridden in particular instantiations of the data in a vector (and the new versions are then selected via the "dynamic method look-up" or "late binding" mechanism). The `Vector` class also implements two (empty) JAVA interfaces, namely `Cloneable` and `Serializable`.

The following JAVA classes are used in the `Vector` class, in one way or another: `ArrayIndexOutOfBoundsException`, `CloneNotSupportedException`, `InternalError`, `Object`, `StringBuffer`, `String`, `System` (all from the `java.lang` package), `Enumeration`, `NoSuchElementException` (both from the `java.util` package), and `Serializable` (from the `java.io` package). These additional classes are relevant for the verification, since they also have to be translated into PVS. They are intertwined via mutual recursion.

7.1.2 Translation of Vector into PVS

The LOOP tool translates JAVA classes into logical theories for PVS, according to the semantics as described before. In this section some aspects of the actual translation of the `Vector` class

– JAVA

```
public class Vector implements Cloneable, java.io.Serializable {
   // fields
   protected Object elementData[];
   protected int elementCount;
   protected int capacityIncrement;

   // constructors
   public Vector(int initialCapacity, int capacityIncrement);
   public Vector(int initialCapacity);
   public Vector();

   // methods
   public final synchronized void copyInto(Object anArray[]);
   public final synchronized void trimToSize();
   public final synchronized void ensureCapacity
                                        (int minCapacity);
   private void ensureCapacityHelper(int minCapacity);
   public final synchronized void setSize(int newSize);
   public final int capacity();
   public final int size();
   public final boolean isEmpty();
   public final synchronized Enumeration elements();
   public final boolean contains(Object elem);
   public final int indexOf(Object elem);
   public final synchronized int indexOf(Object elem, int index);
   public final int lastIndexOf(Object elem);
   public final synchronized int lastIndexOf(Object elem, int index);
   public final synchronized Object elementAt(int index);
   public final synchronized Object firstElement();
   public final synchronized Object lastElement();
   public final synchronized void setElementAt(Object obj, int index);
   public final synchronized void removeElementAt(int index);
   public final synchronized void insertElementAt(Object obj, int index);
   public final synchronized void addElement(Object obj);
   public final synchronized boolean removeElement(Object obj);
   public final synchronized void removeAllElements();
   public synchronized Object clone();
   public final synchronized String toString();
}
```

Figure 7.1: The interface of Java's `Vector` class

are briefly discussed. For this project, it is not needed to translate the whole JAVA library. Only those classes that are actually used in the `Vector` class – called the "surrounding" classes – have to be translated. A further reduction has been applied: from these surrounding classes, only those methods that are actually needed have been translated. Thus, 10K of JAVA code remains, excluding documentation. The LOOP tool turns it into about 500K of PVS code[3].

JAVA's `Object` and `System` classes have several native methods. A native method lets a programmer use some already existing (non-JAVA) code, by invoking it from within JAVA. In the `Vector` class two native methods are used, namely `clone` from `Object`, and `arraycopy` from `System`. Our own PVS code has been inserted as translation of the method bodies of these native methods. An alternative approach would be to use requirements for these methods, like for `toString` and `equals`, see the next section.

The current version of our LOOP tool handles practically all of "sequential" JAVA, *i.e.* of JAVA without threads. The possible use of vectors in a concurrent scenario is not relevant for this invariant verification. The `synchronized` keyword in the method declarations is therefore simply ignored.

There is one point where we have cheated a bit in the `Vector` translation. Often in the `Vector` class an exception is thrown with a message, like in the following code fragment.

– JAVA

```
public final synchronized Object elementAt
                                        (int index) {
  if (index >= elementCount) {
    throw new ArrayIndexOutOfBoundsException
                (index + " >= " + elementCount); }
  ...
}
```

Implicitly in JAVA, the integers `index` and `elementCount` are converted to strings in the exception message. Such conversion is not available in PVS. Of course it can be defined, but that is cumbersome and totally irrelevant for the invariant. Therefore, we have eliminated such exception messages in `throw` clauses, thereby avoiding the conversion issue altogether. This affects the output, but not the invariant.

7.1.3 The class invariant

The first step is to formulate the desired class invariant property. Finding an appropriate, provable, invariant is in general a non-trivial exercise. Usually one starts with some desired property, but to be able to prove that this is an invariant, it has to be strengthened in an appropriate manner[4]. As suggested by the informal documentation in the `Vector` class, a class invariant could be:

> the number of elements in the array of a vector object never exceeds its capacity.

[3]This may seem a formidable size multiplication, but it does not present problems in verification. Reductions in size may still be possible by making more efficient use of parametrisation in PVS code generation.

[4]This is in analogy with "induction loading", where a statement that one wishes to prove by induction must be strengthened in order to get the induction going.

– JML

```
/*@ public invariant:
  @   elementData != null  &&
  @   elementCount <= elementData.length  && // main point
  @   elementCount >= 0  &&
  @   elementData != this  &&
  @   elementData instanceof Object[]  &&
  @   (\forall (int i)
  @       0 <= i && i < elementData.length
  @       ==> (elementData[i] == null ||
  @            elementData[i] instanceof Object));
  @*/
```

Figure 7.2: Main ingredients of invariant of class `Vector`

This property alone can not be proven to be a class invariant. Strengthening is necessary to obtain an actual invariant. This invariant has been obtained "by hand", and not via some form of automatic invariant generation. Precisely annotating all the methods in `Vector` with JML-specifications helps in finding the appropriate strengthening, because it brings forward the pre-conditions for normal and abrupt terminations. The strengthened version of the above property can be extracted from these pre-conditions for normal termination. During verification it turned out that the resulting property had to be strengthened only once more (in a very subtle manner). The main ingredients of the invariant are stated in JML in Figure 7.2.

One more requirement is needed that is directly related to the particular memory model that we use (see Section 2.5), and is not expressible in JML. It says that `elementData` refers to an "allocated" cell in the heap memory, whose position is below the heaptop.

The resulting combined property on OM will be called VectorIntegrity?. Notice that this property says nothing about the value of the `capacityIncrement` field. One would expect that this field should be positive, but this is not the case, because the only time `capacityIncrement` is actually used (in the body of the method `ensureCapacityHelper`), it is first tested whether its value is greater than zero. The informal documentation for this field states that "if the capacity increment is 0, the capacity of the vector is doubled each time it needs to grow", but a more precise statement would be "if the capacity increment is 0 *or less*, ...".

7.1.4 Verification of the class invariant of Vector

After translation of the `Vector` class (and all surrounding classes), the generated theories are loaded into PVS and the verification effort starts. This means that we have to show that the predicate VectorIntegrity? is indeed an invariant. To this end, it has to be shown that (1) VectorIntegrity? is established by the constructors and (2) that VectorIntegrity? is preserved by all public methods of class `Vector`, see Sections 2.6.3 and 6.1.3. Notice that it is essential that the fields of the `Vector` class are protected, so that they cannot be accessed directly from the outside, and the VectorIntegrity? predicate cannot be corrupted in this manner.

Before going into some proof details, we illustrate that detecting all possible exceptions is a non-trivial, but useful exercise. Therefore we consider the following fragment from the `Vec-`

tor class, which describes the method copyInto together with its informal documentation.

– JAVA
```
/**
 * Copies the components of this vector into the
 * specified array. The array must be big enough to
 * hold all the objects in this vector.
 *
 * @param   anArray   the array into which the components
 *                    get copied.
 * @since   JDK1.0
 */
public final synchronized void copyInto
                                    (Object anArray[]) {
    int i = elementCount;
    while (i-- > 0) {
        anArray[i] = elementData[i];
    }
}
```

This method throws an exception in each of the following cases.

- The field elementCount is greater than zero, and the argument array anArray is a null reference;
- elementCount is greater than zero, anArray is a non-null reference, and its length is less than elementCount;
- elementCount is greater than zero, anArray is a non-null reference, its length is at least elementCount, and there is an index i below elementCount such that the (run-time) class of elementData[i] is not assignment compatible with the (run-time) class of anArray.

The first of these three cases produces a NullPointerException, the second one an ArrayIndexOutOfBoundsException, the third one an ArrayStoreException[5]. This last case is subtle, and not documented at all; it can easily be overlooked. But in all three cases, no data in Vector is corrupted, and the predicate VectorIntegrity? still holds in the resulting (abnormal) state.

Below the verification in PVS of several methods is discussed in some detail, namely of setElementAt, toString and indexOf. These methods are exemplaric: the method setElementAt is a typical example of a method for which the invariant is verified automatically (by rewriting). The verification of toString shows how we deal with late binding and indexOf demonstrates the use of the extended Hoare logic for JAVA. The verifications make

[5]See the explanation in [GJSB00], Subsection 15.25.1, second paragraph on page 371. This exception occurs for example during execution of the following (compilable, but silly) code fragment.

```
Vector v = new Vector();
v.addElement(new Object());
v.copyInto(new Integer[1]);
```

extensive use of automatic rewriting to increase the level of automation. For instance, the low-level memory manipulations (involving the get- and put-operations from Section 2.5) require no user interaction at all. Automatic rewriting is also very useful in verifications using Hoare logic, because it simplifies the application of the rules.

Verification of setElementAt

The first method that is discussed in more detail is `setElementAt`. This method takes a parameter `obj` belonging to class `Object` and an integer `index`, and replaces the element at position `index` in the vector with `obj`. A possible JML specification for this method looks as follows.

– JML

```
/*@
  @ normal_behavior
  @   requires: index >= 0 && index < elementCount;
  @   ensures:
  @     (\forall (int i) 0 <= i && i < elementCount ==>
  @        ((i == index && elementData[i] == obj) ||
  @         (i != index && elementData[i] ==
  @                          \old(elementData[i]))));
  @ also
  @ exceptional_behavior
  @   requires: index < 0 || index >= elementCount;
  @   signals: (ArrayIndexOutOfBoundsException)
  @     (\forall (int i) 0 <= i && i < elementCount ==>
  @        elementData[i] == \old(elementData[i]));
  @*/
public final synchronized void setElementAt
                                (Object obj, int index) {
  if (index >= elementCount) {
     throw new ArrayIndexOutOfBoundsException
                    (index + " >= " + elementCount);
    }
  elementData[index] = obj;
}
```

Notice that we have given a "functional" specification by describing post-conditions for this method. These post-conditions can be strengthened further, *e.g.* by including that the fields `elementCount` and `capacityIncrement` are not changed. But for our invariant verification, these post-conditions are usually not relevant, and so we shall simply write `true` in the `ensures:` clause, giving so-called lightweight specifications (like in [PBJ00]). In contrast, the pre-conditions are highly relevant.

Ignoring the post-conditions, the proof obligations (see Section 6.2) for this method are:

∀obj: RefType. ∀index: int.
[λx: OM. VectorIntegrity? x ∧ index ≥ 0 ∧ index < elementCount x]
setElementAt(obj, index)
[VectorIntegrity?]

∀obj: RefType. ∀index: int.
[λx: OM. VectorIntegrity? x ∧ index < 0 ∨ index ≥ elementCount x]
setElementAt(obj, index)
[exception(VectorIntegrity?, "ArrayIndexOutOfBoundsException")]

The proofs of these properties proceed mainly by automatic rewriting in PVS. For the first proof obligation, regarding normal termination, we do explicitly have to make the case distinction whether the argument obj is a reference not.

Verification of toString

Unfortunately, the correctness of the methods in `Vector` is not always as easy to prove as for the above example `setElementAt`. Several methods in the `Vector` class invoke other methods, or contain `while` or `for` loops. Above, we already have seen `copyInto` as an example of such a method. We now concentrate on the method invocations in `Vector`'s `toString` method.

Recall that each class in JAVA inherits the `toString` method from the root class `Object`. In a specific class this method is usually overridden to give a suitable string representation for objects of that class. For a vector object the `toString` method in the `Vector` class yields a string representation of the form $[s_0, \ldots, s_{n-1}]$, where n is the vector's size `elementCount`, and s_i is the string obtained by applying the `toString` method to the ith element in the vector's array. The particular implementations that get executed as a result of these `toString` invocations are determined by the actual (run-time) types of the elements in the array (via the late binding mechanism). Thus they cannot be determined statically (see also Section 6.4).

The annotated JAVA code of `toString` in `Vector` looks as follows.

– JML

```
/*@
  @ normal_behavior
  @   requires: (\forall (int i) 0 <= i && i < elementCount
  @                   ==> elementData[i] != null);
  @   ensures: true;
  @ also
  @ exceptional_behavior
  @   requires: elementCount > 0 &&
  @             !(\forall (int i) 0 <= i && i < elementCount
  @                   ==> elementData[i] != null);
  @   signals: (NullPointerException) true;
  @*/
public final synchronized String toString() {
```

```
    int max = size() - 1;
    StringBuffer buf = new StringBuffer();
    Enumeration e = elements();
    buf.append("[");
    for (int i = 0 ; i <= max ; i++) {
      String s = e.nextElement().toString();
      buf.append(s);
      if (i < max) {
        buf.append(", "); }}
    buf.append("]");
    return buf.toString();
  }
```

It reveals an undocumented possible source of abrupt termination: when one of the elements of a vector's array is a null reference, invoking `toString` on it yields a `NullPointerException`.

The "behavioural subtyping" approach to late binding that we take here (see [Mey97, LW94, Ame90] and Section 6.4), involves writing down requirements on the method `toString` in `Object` and using these requirements in reasoning. In our verification, we thus assume that the definition of `toString` that is actually used at run-time satisfies these requirements, *i.e.* that it is a behavioural subtype of `toString` in `Object`. Thus, we prove that `toString` in `Vector` works correctly, assuming that we have a reasonable implementation of `toString`, without unexpected behaviour.

In ordinary language, the requirements on `toString` say that

- it terminates normally, and has no side-effects;
- it returns a non-null reference to a memory location in newly allocated memory, *i.e.* above the heaptop in the pre-state, but below the heaptop in the post-state (after execution of `toString`);
- this reference has run-time type `String`, and points to a memory cell with integer fields `offset` and `count` (from class `String`), which are non-negative, and an array field `value` (also from `String`), which
  - is a non-null reference with a cell position which is above the heaptop in the pre-state, below the heaptop in the post-state, and different from the previously mentioned `String` reference;
  - has run-time elementtype `char` and a length exceeding the sum of `offset` and `count`.

The verification of the `toString` method from `Vector` is then not difficult, but very laborious. This is because it uses (indirectly via `append` from `StringBuffer`) several different methods from other classes, like `extendCapacity` from `StringBuffer`, and `getChars`, `valueOf` from `String`. For all these methods appropriate "modifies" results – describing which cells and positions can be modified – are needed to prove that the methods do not affect the VectorIntegrity? predicate.

Verification of indexOf

Next we consider the verification of a `for` loop, namely in the method `indexOf`. This verification makes extensive use of the Hoare logic rules as described in Chapter 5.

First consider the specification and implementation of `indexOf`.

– JML

```
/*@
  @ normal_behavior
  @   requires: index >= elementCount ||
  @              (elem != null && index >= 0);
  @   ensures: true;
  @ also
  @ exceptional_behavior
  @   requires: elem == null && index < elementCount;
  @   signals: (NullPointerException) true;
  @ also
  @ exceptional_behavior
  @   requires: elem != null && index < 0;
  @   signals: (ArrayIndexOutOfBoundsException) true;
  @*/
public final synchronized int indexOf
                                  (Object elem, int index) {
    for (int i = index ; i < elementCount ; i++) {
        if (elem.equals(elementData[i])) {
            return i;
        }
    }
    return -1;
}
```

The method `indexOf` takes a parameter `elem` belonging to class `Object` and an integer parameter `index`, and checks whether `elem` occurs in the segment of the vector between `index` and `elementCount`. If so, the position at which it occurs is returned, otherwise -1 is returned.

Notice that the `equals` method in the condition of the `if` statement is invoked on the parameter `elem`. Since we cannot know `elem`'s run-time type, we also have to use the behavioural subtype approach here, and assume that certain requirements hold for `equals`, like for `toString` in the previous example. We shall not elaborate on this point, but concentrate on the `for` loop.

To show that `indexOf` maintains VectorIntegrity?, several cases are distinguished. If the parameter `elem` is non-null and `index` is non-negative, the Hoare logic rules for abruptly terminating loops, as described in Chapter 5, are needed for the verification. A distinction is made between the case that `elem` is found, and that it is not found (because in the first case the for loop terminates abruptly, because of a return, and in the second case it terminates normally, thus different rules have to be used). In both cases it is shown that the method preserves VectorIntegrity?. To this end, the following rule for total return correctness of a `for` loop, is used.

l	bot
C	$[\![$`i < elementCount`$]\!]$
U	$[\![$`i++`$]\!]$
S	$[\![$`if (elem.equals(elementData[i])) {return i;}`$]\!]$
variant	$[\![$`elementCount - i`$]\!]$
P	λx : OM. VectorIntegrity? $x \wedge$ `i` $\geq$ `index` $\wedge$ `i` $\leq$ `elementCount` $\wedge$ $(\exists j.$ `index` $\leq j <$ `elementCount` $\wedge\ j \geq$ `i` $\wedge$ `elem.equals(elementData[`j`]))` $\wedge$ $(\forall k.$ `index` $\leq k <$ `i` $\supset$ $\neg$`elem.equals(elementData[`k`]))`
Q	VectorIntegrity?

Figure 7.3: Instantiation of the total return FOR rule for verification of `indexOf`

– TYPE THEORY

$$\frac{\begin{array}{c}\text{well\_founded?}(R) \\ [P]\ \text{CATCH-STAT-RETURN}(\text{E2S}(C)\,;\,\text{CATCH-CONTINUE}(l\!l)(S)\,;\,U)\ [\text{true}] \\ \forall a.\ \{P \wedge \text{true}(C) \wedge \mathit{variant} = a\} \\ \quad \text{E2S}(C)\,;\,\text{CATCH-CONTINUE}(l\!l)(S)\,;\,U \\ \quad \{P \wedge \text{true}(C) \wedge (\mathit{variant}, a) \in R\} \\ \{P\}\ \text{E2S}(C)\,;\,\text{CATCH-CONTINUE}(l\!l)(S)\,;\,U\ \{\text{return}(Q)\}\end{array}}{[P]\ \text{FOR}(l\!l)(C)(U)(S)\ [\text{return}(Q)]}$$

Notice the similarity with the rule for total break correctness of the `while` statement, as described in Section 5.4. The main difference is that the `for` loop has a different iteration body, namely $\text{E2S}(C)\,;\,\text{CATCH-CONTINUE}(l)(S)\,;\,U$, where U is the formalisation of the update statement of the `for` loop. Recall that for `while` loops the iteration body is $\text{E2S}(C)\,;\,\text{CATCH-CONTINUE}(l)(S)$.

The instantiation of this rule is depicted in Figure 7.3. Notice that the loop invariant (P) implies that the condition `i < elementCount` remains true, because if `i` would be equal to `elementCount`, the last two clauses of the invariant would be contradicting.

In the case that `elem` is not found in the vector, the rule for total (normal) correctness of the `for` loop is used, with a similar instantiation, to show that in that case the loop always terminates normally (returning -1).

In the case that `index` $\geq$ `elementCount`, or in the case of abrupt termination (*i.e.* when `index` < 0 or `elem` is a null pointer), it can be shown that the condition of the `for`-loop immediately evaluates to false or throws an exception, respectively. Since no changes are made to the fields of `Vector`, the property VectorIntegrity? is preserved.

Actually we have proved a bit more about the `indexOf` method than stated here. More is needed because the method is used in another `Vector` method, namely `contains`. With these stronger results, the `contains` method can be verified by automatic rewriting in PVS. In this case late binding cannot occur because the `indexOf` method is declared as `final`, so that it cannot be overridden.

7.1.5 Conclusions and experiences

We have formally proved with PVS a non-trivial safety property for the `Vector` class from JAVA's standard library. The verification is based on careful (lightweight) specifications of all `Vector` methods, using the experimental behavioural interface specification language JML. It makes many non-trivial and poorly documented (normal and abnormal) termination conditions explicit, see also [Vec].

The whole invariant verification presented here was a lot of work. In total, it involved 13,193 proof commands (atomic interactions) in PVS. Some methods required only a few proof commands – and could be verified entirely by automatic rewriting – but others required more interaction. The `toString` method was most labour intensive, requiring 4,922 proof commands, about one third of the total number. Quantifying the time it took is more difficult, because much of the work was done for the first time in such a large project, and could be done faster given more experience. But 3-4 months full-time work (for a single, experienced person) seems a reasonable estimate.

Recall from Subsection 2.3 that our semantics has many output options for statements and expressions. All these possibilities have to be considered in each method invocation. A proof tool is thus indispensable, because it relentlessly keeps track of all options: it happened several times that half-way a proof in PVS a subtle omission in a pre-condition became apparent. Of course, using a proof tool also gives considerable overhead, especially in cases which are obvious to humans. But still, in our experience, it is rewarding to use a proof tool also in such cases, because it is so easy to overlook a detail and make a small mistake. It is in general important to achieve a high level of automation via appropriate rewrite lemmas (as in our semantics) and powerful decision procedures (as incorporated in PVS). Still, substantial performance improvements of proof tools (and the underlying hardware) are highly desirable.

7.2 Verification of Java's AbstractCollection class in Isabelle

This second case study describes a verification of the functional specifications of the methods in the class `AbstractCollection` in JAVA's standard library[6]. The functional specification (or pre-post-condition relation) of a method describes a methods behaviour, *i.e.* how a method changes the state of an object and what the result of the method is.

[6]We use version number 1.25, written by Josh Block, under Sun Microsystems copyright from the JDK1.2 distribution. The implementation of `Vector` in this distribution forms part of the collection hierarchy and is different from the implementation of `Vector` used in the previous case study, mainly because it supports extra operations that are declared in the collection hierarchy.

The JAVA standard library contains several collection or container classes, like `Set` and `List` which can be used to store objects. These collection classes form a hierarchy, with the interface `Collection` as root. This interface declares all the basic operations on collections, such as `add`, `remove`, `size` *etc.*

– JAVA

```
public interface Collection {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator iterator();
    Object[] toArray();
    Object[] toArray(Object a[]);
    boolean add(Object o);
    boolean remove(Object o);
    boolean containsAll(Collection c);
    boolean addAll(Collection c);
    boolean removeAll(Collection c);
    boolean retainAll(Collection c);
    void clear();
    boolean equals(Object o);
    int hashCode();
}
```

The method `iterator` in this interface returns an object implementing the `Iterator` interface. Iterators are intended to provide a way to visit all the elements in the collection.

– JAVA

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

The `Collection` interface declares a method with run-time type `Iterator`. From the methods declared in the `Iterator` interface it seems like `Iterator` does not depend on `Collection`. But, the informal specification [Jav] explains that a mutually recursive dependency is intended, and every iterator has a reference to the collection underlying it. The remove method in the iterator even removes an element from this underlying collection.

A small part of the collection hierarchy is displayed in Figure 7.4. The `Collection` interface is the root of this hierarchy. It contains several subinterfaces, *e.g.* interfaces `List` and `Set`. These interfaces declare the signature of a collection, list, set *etc.* Classes which implement these interfaces have to be provide implementations for these methods. At the bottom in the hierarchy are complete implementations of collection structures, *e.g.* `Vector` and `LinkedList`. These classes can immediately be used by application programmers. The classes in the middle of the hierarchy, such as `AbstractCollection` and `AbstractList` give an incomplete implementation of the interfaces. They contain several methods without an implementation, so-called abstract methods, and the other methods are implemented in terms of

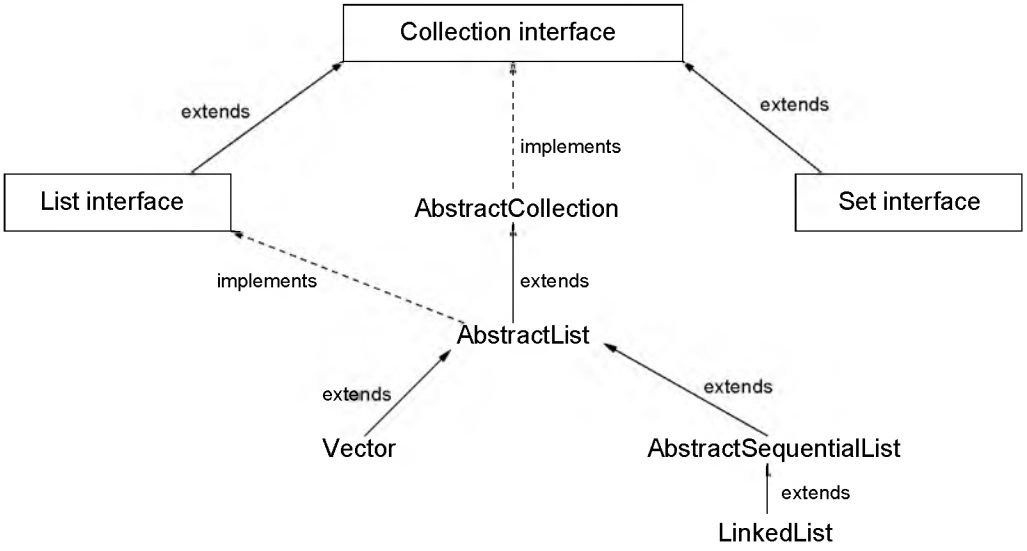

Figure 7.4: Part of the Collection hierarchy

these abstract methods. This gives users of the JAVA library the possibility to program their own classes, by implementing only the abstract methods and inheriting the other implementations. Of course, the other methods may be overridden in subclasses. Since JAVA-1.2, the abstract collection classes also provide so-called *optional methods*. In the abstract class such a method is implemented by throwing an `UnsupportedOperationException`. The programmer of a subclass, which inherits from the abstract class, has the choice whether he wants to provide a different implementation for this method by overriding it. There has been some objection to the introduction of these optional methods in the library classes [Bud00], because users of the library have to be aware of the possibility that the optional methods may be unimplemented.

In this case study, the class `AbstractCollection`, implementing the `Collection` interface, is discussed. This class has abstract methods `size` and `iterator`, and the method `add` throws an `UnsupportedOperationException`, which makes it an optional method. The other methods declared in `Collection` are all implemented in `AbstractCollection` in terms of the methods `size`, `add`, `iterator` and the methods from `Iterator`.

To implement a so-called *unmodifiable collection*, it is sufficient to make a class inherit from `AbstractCollection` and to give implementations for the `size` and `iterator` method. The object that is returned by the `iterator` method should implement the methods `hasNext` and `next` from the interface `Iterator`, the `remove` method may throw an `UnsupportedOperationException`. To implement a so-called *modifiable collection*, additionally the method `add` must be overridden in the subclass, and the object that is returned by the `iterator` method must implement the `remove` method from the class `Iterator` as well. Notice that only because `add` is an optional method, it is possible to make unmodifiable collections by using the `AbstractCollection` class.

To verify the specification of `AbstractCollection` the following approach is taken. Following the informal description in the interfaces `Collection` and `Iterator`, formal specifications for all methods are written in JML, describing their functional behaviour. Subsequently, the methods that are implemented in class `AbstractCollection` are shown to satisfy these specifications from `Collection` (provided that the (abstract) methods that are used in their implementations satisfy their specifications).

The verification is a typical example of a modular verification, where a single module (a class) is verified in isolation, using specifications of the methods from other classes (components, or later to be implemented subclasses), as described in Section 6.4. The specifications of the methods in `Collection` and `Iterator` are discussed, followed by a presentation of the verifications of the implementations in `AbstractCollection`. The contribution of this case study is that it gives a clear (and correct) specification of the methods in a collection. However, even more important is that it applies modular verification in practice, and forces us to deal with all the details of the issues involved.

Section 7.2.1 discusses the JML class specifications of `Collection` and `Iterator`. These are translated by hand into ISABELLE specifications. This translation is discussed in Section 7.2.2. Subsequently, Section 7.2.3 discusses the verification of the method implementations in `AbstractCollection` *w.r.t.* the specifications of the methods in `Collection` and `Iterator`. Finally, Section 7.2.4 concludes and discusses experiences in constructing the specifications and correctness proofs.

7.2.1 The specification of Collection and Iterator

The first step in the actual case study is to write specifications for the methods in the interfaces `Collection` and `Iterator`. For these specifications we will use a JML-like notation (as introduced in Chapter 6). For readability, we sometimes use notations from ISABELLE in the assertions. Since the specifications of `Collection` and `Iterator` are closely connected, we present their class specifications together. First we discuss the model variables used in the specification of `Collection`, then the model variables used in `Iterator`. Then we specify the methods declared in the interface `Iterator`. Subsequently, we specify the methods of `Collection`.

The model variables of Collection

The first step in writing the specifications is to decide how the collection will be modeled. The interface `Collection` itself does not contain any variables (see page 168), but several model variables are used to describe the behaviour of the collection. As explained in Section 6.4, these model variables can be used freely in method specifications. For concrete implementations of `Collection` a representation function, relating its concrete fields to the model variables have to be given. However, in this case study, the only implementation of `Collection` that we consider is the class `AbstractCollection`. This class only gives an abstract implementation and does not declare any fields, thus we do not have to give such a representation function.

As can be seen from the informal specification of `Collection` [Jav], the contents of a collection can be represented as a bag (or multiset) of objects. We use the ISABELLE type `'a multiset` for this model variable. Objects are represented as references, thus the model variable `contents` is declared as follows[7].

– JML

```
/*@ public model (refType' multiset) contents
  @*/
```

[7]Notice that we can declare the model variable with this type because we do this translation by hand. If this translation would have been done by a compiler, we should have declared the variable as *e.g.* `JMLObjectBag`, and provided a mapping from the operations in this pure class to the operations on multisets in ISABELLE.

name	type	represents
`contents`	`refType' multiset`	contents of collection
`addDefined`	`boolean`	true iff `add` operation supported[8]
`removeDefined`	`boolean`	true iff `iterator` returns an object implementing `Iterator` where the `remove` operation is supported
`storable`	`refType' => boolean`	holds for all elements for which `add` operation does not throw an exception
`allowDoubles`	`boolean`	true iff collection can contain same element more than once

Figure 7.5: Model variables used in the specification of interface `Collection`

Some of the JML specifications below contain quantifications over objects. In the translated specifications, *i.e.* the ISABELLE specifications, this is translated into a quantification over elements in `refType'`, plus an assumption that the references satisfy the class specification of `Object`. In our case, this simplifies to an assumption that the object satisfies the specification of the method `equals`.

Further, several model variables are used which deal with choices that are left to implementations of `Collection`, *i.e.* whether the optional methods `add` and `remove` (in the iterator) are implemented, which elements are storable in the collection and whether double elements are allowed in the collection. Figure 7.5 gives an overview of the model variables for the interface `Collection`.

Further, we use a dependency constraint on the model variables which states when they may have changed. The variables `addDefined`, `removeDefined`, `allowDoubles` and `storable` are all constant, thus they have the same value in every state. We assume that the value of `contents` is preserved if the heap is not changed at position p, where p is the memory location where the fields of the collection are stored. Actually, we should have used another model variable `state`, modelling the internal state of `Collection`. In concrete implementations, this `state` would depend on all the fields in the concrete implementation. In the specification of `Collection` we would state that `contents` depends on `state`. Every state change would thus imply a possible change of the contents of the collection. At the moment, the machinery to express exactly what is meant by the state of an object is not available, therefore we choose simply to make `contents` depend on the memory of the heap at position p (the position where the collection is stored). Since the operations on collections only change the pointers to the stored elements, but never change the elements themselves, this is a reasonable simplification: it does not influence correctness.

As an invariant of `Collection` we use the following properties

- `contents` always is a finite bag
- if `allowDoubles` is true, every element occurs at most once in the collection (*w.r.t.* the `equals` operation on these objects)

[8] An operation is supported if its definition is overridden, so that it does not throw an `UnsupportedOperationException` anymore.

name	type	represents
contents	refType' multiset	the elements through which is iterated
removeDefined	boolean	true iff remove operation supported
underlyingCollection	refType'	reference to underlying collection
lastElement	refType'	reference to element last returned by next
removeAllowed	boolean	true iff remove operation will not throw exception

Figure 7.6: Model variables used in the specification of interface Iterator

In most cases this invariant follows redundantly from the specifications. Only in the correctness proof of the method addAll we need to show that the second item is preserved. In the correctness proofs of the other methods we sometimes use that contents is a finite bag.

The model variables of Iterator

The purpose of the Iterator interface (see page 168) is that it provides means to walk through all the elements of a collection, and possibly remove the element that has just been visited from the underlying collection. Thus, the iterator is closely connected to the underlying collection. Again, the interface does not declare any variables, but several model variables are used to write the specifications. Figure 7.6 gives an overview of the model variables used in the specification of Iterator.

The model variable contents initially contains the elements of the collection that is iterated through. During iteration, every visited element is removed from this collection, thus ensuring that every element is visited exactly once. Just as the model variable contents in Collection, this model variable has type refType' multiset, where the references in this multiset are instances of class Object.

The remove operation in the Iterator interface is optional, to implement an unmodifiable collection, an implementor can make this method throw an UnsupportedOperationException. Whether this is the case is denoted by the model variable removeDefined.

The model variable underlyingCollection maintains a reference to the collection that constructs the object implementing Iterator. The remove method declared in Iterator removes an element from the underlying collection.

Every remove operation has to be preceded by one or more next operations (possibly with a number of hasNext operations in between). The remove operation removes the value that was returned by the last next operation. Thus, after a remove has been done, a new next operation has to be applied first, before another remove is allowed. Whether a remove is allowed is denoted by the variable removeAllowed. The value that will be actually removed is maintained in lastElement.

The model variables underlyingCollection and removeDefined are constant, the values of removeAllowed, lastElement and contents are preserved as long as the heap is not changed at the position of the iterator object (thus they depend on the state of the

iterator).

As an invariant of `Iterator` we specify that `contents` is a finite bag.

The specification of the methods in Iterator

The next step is to give specifications for the methods in the `Iterator` interface. As mentioned above, the `Iterator` interface (see page 168) declares three methods: `hasNext()`, `next()` and `remove()`. The method `remove()` is an optional method, to implement an unmodifiable collection it does not have to be supported. Below, we discuss the specification for each of these methods.

`hasNext()` This operation checks whether there are still elements that have not been visited yet. It always terminates normally and does not have side-effects. In this specification we use the ISABELLE notation {#} to denote the empty bag.

– JML

```
/*@ normal_behavior
  @   ensures: \result == (contents != {#});
 @*/
public boolean hasNext();
```

`next()` This operation returns an element from `contents` of `Iterator`. Every element should be visited only once, therefore the returned element is also removed from the `contents` of the iterator. It is unspecified which element is returned[9]. The `next` operation only terminates normally if the `contents` are not empty. Besides changing the value of `contents`, this method also sets the values of `lastElement` and `removeAllowed` appropriately. We only use the normal behaviour specification of this method, because in the `AbstractCollection` class the `next` method is never called without checking `hasNext`.

– JML

```
/*@ normal_behavior
  @     requires: contents != {#};
  @   modifiable: contents, lastElement, removeAllowed;
  @      ensures: contents ==
  @                  \old(contents) - {#\result#} &&
  @               \old(contents.elem(\result) &&
  @               removeAllowed &&
  @               lastElement == \result;
  @*/
public Object next();
```

[9]Here we actually cheat a bit: according to the specification, if the elements in the collection would be returned by the iterator in some specific order, they would be stored according to this order in the resulting array. To specify this would require an extra model variable `R` representing the order. The `next` operation would return elements *w.r.t.* this order. Thus, restrictions on the order would be necessary to ensure that it is always known which element will be returned by the `next` operation. Leaving this out implies that the method `toArray` could not be specified completely.

The - operation is the remove (or difference) operation on bags in ISABELLE. A singleton bag containing the element v is denoted as {#v#}.

remove() The last method that is declared in the `Iterator` interface is `remove`. This method only terminates normally if the `remove` operation is supported and if there has been a call to `next` before (denoted by `removeAllowed`). If so, it removes one occurrence of the element that was returned by the last `next` from the collection underlying the `Iterator`. Thus, for example after three invocations of `next`, `remove` can be invoked only once. Its specification is as follows.

– JML

```
/*@ normal_behavior
  @   requires: removeDefined && removeAllowed;
  @ modifiable: underlyingCollection.contents, removeAllowed;
  @    ensures: underlyingCollection.contents ==
  @               \old(underlyingCollection.contents) -
  @               {#lastElement#} &&
  @             !removeAllowed;
  @*/
public void remove();
```

Specifications of the methods of Collection

The last step in writing the specification is to make specifications for the methods in `Collection`.

First we discuss the specifications of the methods that are abstract or unsupported in `AbstractCollection`. These specifications are based on the informal specifications [Jav] only.

size() The `size` method returns the number of elements in the collection (or, if the collection is too big `integer.MAX_VALUE`). The method always terminates normally, and does not have any side-effects.

– JML

```
/*@ normal_behavior
  @   ensures: \result == min(size (contents),
  @                           integer.MAX_VALUE);
  @*/
public abstract int size();
```

`iterator()` This method returns an instance of a class correctly implementing the `Iterator` interface. Thus, the result can not be a null-reference. Following the behavioural subtype approach, this follows from specifying that the result should be an instance of `Iterator`. Further, we ensure that the `iterator` is initialised correctly by specifying the initial values of its model variables. By specifying that `iterator` has no side-effects, we require that the `Iterator` is created in a newly allocated memory cell, *i.e.* above the old heaptop. As explained in Section 6.5 a method is considered to have side-effects if it changes memory that was allocated already before the method call, thus a method without side-effects is allowed to allocate new memory. The `iterator` method always terminates normally.

– JML

```
/*@ normal_behavior
  @     ensures: \result instanceof Iterator &&
  @              \result.contents == contents &&
  @              \result.removeDefined == removeDefined &&
  @              \result.underlyingCollection == this &&
  @              !\result.removeAllowed;
  @*/
public Iterator iterator();
```

`add(Object o)` The last method for which no (sensible) implementation is given in `AbstractCollection`[10] is `add`. This method only terminates normally if the collection is modifiable (and thus the `add` operation has been overridden), and if the parameter object is storable in the collection. According to the documentation, a particular implementation might refuse to add certain objects, for example it might refuse to store `null` references. Abstractly, this is specified by the predicate `storable` (see Figure 7.5). If an object is not storable, the `add` method will not terminate normally.

If the parameter object is storable in the collection, it still might be the case that it already occurs in the collection and that the collection does not allow elements to be stored twice. Then, the element is not added, and the method returns `false`. Otherwise, the element is added and `true` is returned. Before writing this specification, it should be discussed what it means that an element already occurs. Testing whether an element already occurs can not be done by using pointer equality, because two different non-null references might be considered equal by a particular `equals` implementation (which overrides the definition of `equals` in `Object`). However, comparing two null-references really requires testing pointer equality. Therefore we introduce the following abbreviation which tests for occurrence of an element *w.r.t.* the `equals` operation for non-null references, where `elem` is the ISABELLE test for occurrence of an element in a bag. This operation is an operation on multisets. Formally, we would have to define it in a pure class like `JMLObjectBag`.

[10]Remember that `add` is implemented in `AbstractCollection` by throwing an `UnsupportedOperationException`.

– JML

```
/*@ model boolean occurs(Object o) {
  @   return (o == null ?
  @           elem(null) :
  @           (\exists (Object x) elem(x) && o.equals(x)));
  @ }
  @*/
```

Using this abbreviation the `add` method is specified as follows.

– JML

```
/*@ normal_behavior
  @     requires: addDefined && storable(o);
  @   modifiable: contents;
  @      ensures: \result == (contents != \old(contents)) &&
  @               (!allowDoubles &&
  @                 \old(contents.occurs(o))) ?
  @                contents == \old(contents) :
  @                contents == \old(contents) + {#o#};
  @*/
public boolean add(Object o);
```

For the specifications of the other methods in `Collection`, *i.e.* the methods that have an implementation in `AbstractCollection`, we look both at their informal specification (in `Collection`) and their implementation (in `AbstractCollection`). Many of the specifications are similar, therefore only several exemplaric specifications (and verifications later) are discussed.

`isEmpty()` The specification of `isEmpty` is straightforward: it simply tests whether the collection is empty and does not have a precondition or side-effects.

– JML

```
/*@ normal_behavior
  @   ensures: \result == (size (contents) == 0);
  @*/
public boolean isEmpty();
```

`remove(Object o)` This `remove` operation invokes the method `remove` from the `Iterator` interface. This method is an optional method, thus it does not have to be supported by implementations of `Iterator`. In that case, the method `remove` from `AbstractCollection` will also throw an `UnsupportedOperationException`. Whether the `remove` operation in `Iterator` is supported is denoted by the model variable `removeDefined`.

The method remove changes the contents of the collection, by testing whether the element occurs, and if so, removing it once. It returns a boolean value which is true if the collection has changed. Notice that we can not simply write contents == \old(*contents*) - {#o#}, because the collection might not contain a reference to o, but a reference to an equal object. The remove operation will then remove this equivalent element, but the multiset difference operator would ignore this equality. To be able to count how many times an element occurs *w.r.t.* the equals operation, we define the following function count_occurs. Just like the model method occurs this method is defined on multisets and formally, we would have to define it in a pure class like JMLObjectBag.

– JML

```
/*@ model int count_occurs(Object o) {
  @   return (o == null ?
  @           count(null) :
  @           setsum (count)
  @                  {x. x : set_of (this) && o.equals(x)});
  @}
  @*/
```

First a set is constructed, containing all the elements in the collection that are equal to o, and subsequently for each of these elements the occurrences are counted. The sum of this is returned by the method.

As the postcondition of remove we want to state after the remove operation, at most one object equal to o is removed. Thus, for every element x equal to o, the number of occurrences decreases by 1 (with 0 as minimum). If x is not equal to o, the number of occurrences is not changed.

However, we need an extra restriction, before we are able to prove this. Suppose that we have the following JAVA class.

– JAVA

```
class RemoveCollectionFromCollection {

  Vector w;

  boolean remove_one_element () {
    Vector v = new Vector ();
    Object o = new Object ();
    v.add(o);
    v.add(v);
    w = (Vector)v.clone();
    boolean first_time = v.contains(w);
    v.remove(o);
    boolean second_times = v.contains(w);
    return (first_time == second_time);
  }
}
```

The method remove_one_element returns false, because after the removal of o, the value

of v has changed and it is not equal to w anymore. In the case that a collection contains itself, it becomes very hard to specify the postcondition of the `remove` operation, therefore in the specification of `remove` we assume that a collection does not contain itself[11]. For similar reasons, in the postcondition we only quantify over objects that are not the collection itself. That this non-trivial condition is necessary to prove the correctness of `remove` only becomes clear during the verification.

– JML

```
/*@ normal_behavior
  @     requires: removeDefined &&
  @                (\forall (Object x)
  @                  (contents.elem(x) ==> x != this));
  @   modifiable: contents;
  @      ensures:
  @        \result == (contents != \old(contents)) &&
  @        (\forall (Object x)
  @            x != this ==>
  @            contents.count_occurs(x) ==
  @            (o == null && x == null) |
  @            (o != null && o.equals(x)) ?
  @            min (\old(contents.count_occurs(x) - 1), 0) :
  @            \old(contents.count_occurs(x)));
  @*/
public boolean remove(Object o);
```

Notice that it can be proven – using the symmetry and transitivity of the equality operation – that the size (*i.e.* the sum of all the counts) of the collection decreases by at most 1.

`addAll(Collection c)` The last method specification that we discuss is the specification of `addAll`. If the collection allows elements to be stored more than once, this method is the same as multiset union, otherwise it adds those elements that do not occur yet. In that case, every element occurs at most once. For this method, we explicitly show that if double elements are not allowed in the collection, this is preserved by this method. For similar reasons as for `remove` above, we assume that both collections do not contain references to `this`.

– JML

```
/*@ normal_behavior
  @     requires: addDefined && c != null && c != this &&
  @               !allowDoubles ==>
  @                  (\forall (Object x)
  @                  (contents.elem(x) |
  @                   (c.contents).elem(x)) ==>
  @                     x != this);
  @               (\forall (Object o)
  @                   (c.contents).elem(o) ==>
```

[11] Actually, we want to state that if the elements in the collection are not affected by changes to the collection structure itself.

```
  @                                   storable(o)) &&
  @                !allowDoubles ==>
  @                    (\forall (Object o)
  @                       contents.occurs(o) <= 1);
  @    modifiable: contents;
  @       ensures:
  @         \result == (contents != \old(contents)) &&
  @         allowDoubles?
  @          contents == \old(contents) + c.contents :
  @          (\forall (Object o)
  @             o != this ==>
  @             (contents.occurs(o) =
  @              (c.contents + \old(contents)).occurs(o)) &&
  @             contents.count_occurs(o) <= 1);
  @*/
public boolean addAll(Collection c);
```

The method `addAll` only terminates normally if the `add` operation is overridden, the argument collection is not a null reference and all elements are storable. Further, as can be seen from the informal specification, its behaviour is unspecified if the argument collection is equal to the current collection. Thus, our specification only specifies the behaviour for `c != this`. The `addAll` operation only modifies the `contents` of the current collection, the contents of the argument are unchanged. It returns true iff the current collection has been changed. If the collection allows elements to be stored more than once, the new collection is exactly the multiset union of the old collection and the argument collection. Otherwise, all the elements that occur in the new collection occurred either in the old collection or in the argument collection, and every element occurs at most once (*w.r.t.* the appropriate `equals` operator).

7.2.2 Translating the specifications into Isabelle

The next step is to translate the JML specifications into the specification language of ISABELLE. At the moment, the LOOP tool is being extended to do this translation automatically, but in this case study the translation is still done by hand. This means that we have to do a bit more work ourselves, but makes no difference for the issues involved. First of all, this translation requires making some aspects of our formalisation explicit, *e.g.* in preconditions it is explicitly stated that the receiving object is in allocated memory: if the contents of the object are stored at memory location p, then $p <$ heaptop x. Further, for every argument, we assume that if it is a reference, its type is a subclass of the declared type. This is ensured by the JAVA compiler, so we can safely assume it. Also, for every argument and every reference type used in the specification, we assume that it satisfies the class specification of its declared type. Thus, for example, everywhere where we quantify over all objects (in the collection), we assume that these objects satisfy the specification of `Object`, thus in particular that they satisfy the specification of the `equals` operation. This is in line with the behavioural subtyping approach (see Chapter 6).

As explained above, for the non-constant model variables in `Collection` and `Iterator` we assume that they may change if the contents of the heap at position p changes, where p is the location on the heap where the contents of the collection are stored. Therefore, if a method

– ISABELLE

```
remove'spec :: [OM' => OM' Iterator'IFace,
                MemLoc'] => bool
"remove'spec c p ==
  (let remove =
         java_util_IteratorInterface.remove' c in
  (ALL z.
   total'correctness
    (%x. x = z & it_removeDefined c x &
         removeAllowed c x & p < heap'top x)
    remove
    (%x. ~ removeAllowed c x &
         (let UC_pos = refpos'(underlyingCollection c x);
              UC_clg = Collection'clg (get'type UC_pos x)
                                      UC_pos
          in col_contents UC_clg x =
                col_contents UC_clg z -
                  {# lastElement c z #} &
             (ALL t. t < heap'top z &
                     t ~= UC_pos -->
                          heap'mem x t = heap'mem z t) &
              get'type UC_pos x = get'type UC_pos z &
              get'dimlen UC_pos x = get'dimlen UC_pos z) &
         lastElement c x = lastElement c z &
         it_contents c x = it_contents c z &
         get'type p x = get'type p z &
         get'dimlen p x = get'dimlen p z &
         heap'top z <= heap'top x &
         stack_equality z x &
         static_equality z x)))"
```

Figure 7.7: Specification of method `remove` from `Iterator` in ISABELLE

changes the heap, but the corresponding modifies clause does not contain a particular (non-constant) model variable, we add in the postcondition that this model variable is unchanged.

For example, the JML specification of the method `remove` in `Iterator` (see page 174), is transformed into the ISABELLE specification in Figure 7.7.

The precondition contains a clause `x = z`. In the postcondition the "logical" variable `z` will be used to evaluate the `\old` expressions. Further, the model variable `removeDefined` is prefixed with `it_` to avoid name clashes in ISABELLE with the variable `removeDefined` from `Collection`. The first conjunct of the postcondition expresses that `removeAllowed` no longer holds. Then, it shows how the `contents` of the underlying collection are changed. Again, the prefix `col_` is used to disambiguate the model variables `contents`. The quantification shows how the heap is changed by this call: at memory location `UC_pos` (which is where the underlying collection is stored), the heap is changed, the rest of the (allocated) heap memory is unaffected. Also, we add assertions stating that the type and dimlen entry of the collection have not changed, *i.e.* it is still the same object. The other model variables in `Collection` are constants, so nothing has to be said about their values. However, the `Iterator` interface contains some model variables that are not constants, but that are also not changed by the `remove` operation. To specify this, the unchanged model variables are also mentioned in the postcondition explicitly. That the variables `it_removeDefined`, `underlyingCollection` *etc.* are not changed follows from the fact that they are constant. Therefore we do not write them explicitly in our specification. The last two conjuncts state that the stack and static memories are unchanged.

Another aspect that is implicit in the JML specification, but explicit in the ISABELLE specification is what it exactly means for an object to be an instance of a certain class. Following the behavioural subtype approach, if an object is an instance of a class, it satisfies its specifications, *i.e.* it satisfies the invariant, all the methods satisfy the appropriate method specifications and model variables satisfy their constraints. When a method specification is translated to ISABELLE, this has to be made explicit. For example, the specification of the method `iterator` on page 175 gives rise to the ISABELLE specification in Figure 7.8. In the postcondition, it is stated that a reference is returned in newly allocated memory, *i.e.* between the old and the new heaptop. This reference points to an object which is an instance of `Iterator`, thus it satisfies its method specifications, invariant and the dependency relation which relates the model variables to the heap. Further, the appropriate initialisations are specified and it is stated that this method does not have side-effects (because it does not change the memory that is already allocated before the method call).

Methods with reference parameters are treated in the same way, *i.e.* in the precondition assumptions are made that they are correct instances of the declared class, satisfying the appropriate specifications.

7.2.3 Verification of the methods in AbstractCollection

Finally, the verification effort can begin. Given the method specifications in `Collection` and `Iterator`, the method implementations in class `AbstractCollection` can be verified. The abstract methods and `add` from `AbstractCollection` are assumed to satisfy their specification. We discuss the verifications of the methods `isEmpty()`, `remove(Object o)` and `addAll(Collection c)` in full detail, as these are typical for all the verifications.

– ISABELLE

```
iterator'spec :: [OM' => OM' Collection'IFace,
                  MemLoc'] => bool
"iterator'spec c p ==
  (let iterator =
         java_util_CollectionInterface.iterator' c in
  (ALL z.
    total'expr_correctness
      (%x. x = z & p < heap'top x)
      iterator
      (% x v. (case v of
                 Null' => False
                |Reference' q =>
                   q < heap'top x &
                   heap'top z <= q &
                   (let clg = Iterator'clg (get'type q x) q
                    in
(*                 / *)  hasNext'spec clg q &
(*                / * )  next'spec clg q &
(*Iterator'spec { *)     remove'spec clg q &
(*                \ *)   Iterator'invariant clg q &
(*                 \ *)  Iterator'dependencies clg q &
                         col_contents c x =
                             it_contents clg x &
                         col_removeDefined c x =
                             it_removeDefined clg x &
                         underlyingCollection clg x =
                             Reference' p &
                         ~ removeAllowed clg x)) &
              heap_equality z x &
              stack_equality z x &
              static_equality z x)))"
```

Figure 7.8: Specification of method `iterator` in ISABELLE

The basis for the verification is the Hoare logic presented in Chapter 5. Using the appropriate proof rules, the methods are decomposed in smaller pieces. In many cases this decomposition can be done automatically: ISABELLE gets a collection of Hoare logic proof rules and applies the appropriate one. However, because most of the methods under consideration contain several calls to other methods, still much user interaction is required.

`isEmpty()` The proof for which we achieved the highest degree of automation is the correctness proof of the method `isEmpty`. In `AbstractCollection` this method is implemented as follows.

– JAVA

```
public boolean isEmpty() {
   return size() == 0;
}
```

The correctness proof of `isEmpty` starts by breaking down the method body, until only the call to the method `size` remains. This is done by ISABELLE, applying appropriate proof rules of our Hoare logic automatically. The subgoal that is constructed in this way (by a single proof command) is depicted in Figure 7.9.

Basically, this goal states that the return value will be the result of comparing the outcome of the `size` method with 0, and that there are no side-effects. To prove this subgoal, the specification of `size` is used. Using a method specification in general involves many mechanical steps and a few creative ones, thus the proof construction process could benefit from having appropriate tactics to do this.

`remove(Object o)` For several methods in `AbstractCollection` different cases have to be distinguished. For each of these cases the correctness of the specification is shown. Consider for example the method `remove`, which is implemented as follows.

– JAVA

```
public boolean remove(Object o) {
   Iterator e = iterator();
    if (o==null) {
       while (e.hasNext()) {
           if (e.next()==null) {
               e.remove();
               return true;
           }
        }
   } else {
       while (e.hasNext()) {
           if (o.equals(e.next())) {
               e.remove();
               return true;
           }
       }
   }
```

– ISABELLE

```
Level 3 (1 subgoal)
[| AbstractCollectionAssert' p (c p); size'spec (c p) p;
   AbstractCollection'dependencies (c p) p; clear'stack |]
==> isEmpty'spec (c p) p
 1. !!z ret'isEmpty ret'isEmpty'becomes za.
    [| AbstractCollectionAssert' p (c p); size'spec (c p) p;
       AbstractCollection'dependencies (c p) p; clear'stack |]
    ==> total'expr_correctness
         (%x.
           x = za &
           put'empty'stack (stack'topinc x #-1)
                           (stack'top x - #1) = z &
           p < heap'top x &
           ret'isEmpty =
           get'boolean (Stack' (stack'top x + #-1) #0) &
           ret'isEmpty'becomes =
           put'boolean (Stack' (stack'top x + #-1) #0))
         (java_util_AbstractCollectionInterface.size' (c p))
         (%u ua.
           ret'isEmpty (ret'isEmpty'becomes u (ua=#0)) =
           (size(abs_col_contents (c p)
            (put'empty'stack
             (stack'topinc (ret'isEmpty'becomes u (ua=#0)) #-1)
             (stack'top (ret'isEmpty'becomes u (ua=#0)) - #1)))=#0) &
           heap_equality z
            (put'empty'stack
             (stack'topinc (ret'isEmpty'becomes u (ua=#0)) #-1)
             (stack'top (ret'isEmpty'becomes u (ua=#0)) - #1))&
           stack_equality z
            (put'empty'stack
             (stack'topinc (ret'isEmpty'becomes u (ua=#0)) #-1)
             (stack'top (ret'isEmpty'becomes u (ua=#0)) - #1))&
           static_equality z
            (put'empty'stack
             (stack'topinc (ret'isEmpty'becomes u (ua=#0)) #-1)
             (stack'top (ret'isEmpty'becomes u (ua=#0)) - #1)))
```

Figure 7.9: Subgoal in correctness proof of method `isEmpty`

```
    return false;
}
```

In the verification of this method different combinations of the following cases have to be distinguished.

- The collection is empty. In that case, the search stops immediately and `false` is returned.
- The argument is a null reference. The method body contains two while loops. In this case, the first while loop is selected, which does pointer comparison on objects.
- The argument is a non-null reference. The second while loop is selected, which uses the `equals` operations to compare objects.
- The argument object occurs in the collection. In that case at some point the while loop (either the first one or the second one) will stop abruptly, returning `true`. For this case it is necessary that the collection is not empty (because then the while loop always terminates normally).
- The argument does not occur. The loop will iterate through all the elements in the collection and then exit normally, returning false.

The loop invariant that is used in the verification of this method basically says that the object `o` is not found yet. Depending on whether we assumed that the element occurs or not, we state that it occurs or does not occur in the remaining iteration collection. If the loop body terminates normally, the element is not found and this invariant remains true. As a variant we use the size of the collection that the iterator iterates through, this decreases with every iteration of the loop body because of the `next` operation.

`addAll(Collection c)` Finally, we look at the verification of the method `addAll`. This is a typical example of a verification with an interesting loop invariant. This method is implemented as follows.

– JAVA

```
public boolean addAll(Collection c) {
    boolean modified = false;
    Iterator e = c.iterator();
    while (e.hasNext()) {
        if(add(e.next()))
            modified = true;
    }
    return modified;
}
```

We are only concerned with the functional behaviour of this method, and do not consider causes for abrupt termination. Therefore we assume that all elements in the argument collection are storable in the collection[12]. The loop in this method body always terminates normally. The loop invariant of this method is the following (where `e` is the reference to the iterator).

– JML

```
/*@ loop_invariant:
  @       allowDoubles ?
  @       contents + e.contents ==
  @           \old(contents) + c.contents :
  @       (\forall (Object x)
  @           (contents + e.contents).occurs(x) ==
  @              (c.contents + \old(contents)).occurs(x) &&
  @            contents.count_occurs(x) <= 1);
  @*/
```

Again, the variant is the size of the contents of the iterator.

– JML

```
/*@ variant_function: size(e.contents);
  @*/
```

The correctness proofs for these methods on average contain 250 proof steps. For most methods several cases are distinguished and 2 or even 4 (slightly) different correctness proofs are required. Many of these proof steps are straightforward, thus by a better use of tactics and rewrite strategies for dealing with abstract variables the length of the proofs could significantly be shortened. A big problem in the verification was the memory use of ISABELLE (often over 350 Mb), which caused the machine[13] to spend much time on swapping.

7.2.4 Conclusions and experiences

We have presented a verification of the functional behaviour of several methods in JAVA's library class `AbstractCollection`. This class gives an abstract implementation of the interface `Collection`, which is the root of the collection hierarchy in the JAVA class library. Based on the informal method specifications, JML specifications for all the methods declared in `Collection` are given. To show that `AbstractCollection` correctly implements the interface `Collection`, it has to satisfy these specifications. To do this verification, the method specifications are translated (by hand) to ISABELLE specifications (but the JAVA code is translated by the LOOP tool). Subsequently, the verifications are done in ISABELLE. At the moment, only crucial parts of the case study have been verified in full detail. ISABELLE has a scaling problem: verification of the methods in `AbstractCollection` uses so much memory that

[12]Notice that if not all elements are storable, an exception will be thrown in the middle of the adding process, *i.e.* half way adding all the elements in the argument collection. In that case, not much can be said about the new contents of the collection, only that it is in between the old one and the union of the old one and the argument, *i.e.* `\old(`*contents*`)` $\leq$ `contents` $\leq$ `\old(`*contents*`) + c.contents`, where $\leq$ is the submultiset operator.

[13]The proofs are done on a Pentium II, 300 MHz and a Pentium III, 500 MHz, both with 256 Mb RAM.

it significantly slows down the verification. In particular the performance of the powerful proof commands (like calling the simplifier) is seriously affected by this. The memory usage of ISABELLE is thus a big problem in the verifications, because the computer spends too much time on swapping, which interferes with interactive verification.

This verification is a typical example of a modular verification. It applies the theory of modular verification in practice, which forces us to deal with all the details of the issues involved in modular verification. Reasoning about method invocations is done using the method specifications, instead of the implementations. This is typical for the verification of object-oriented programs, where the binding of method bodies to method calls only can be done at run-time. Because of the concept of an abstract class, the crucial manipulations on collections are all done in the abstract methods. The methods that have been verified all iterate over a collection and invoke methods to change the collection. They are independent of the actual implementation of the collection. In a subclass, the implementation of the abstract methods is closely related with the representation of the collection. Reasoning with the specifications instead of the method implementations makes the method verifications inherently more difficult, than verification of program verification in a traditional imperative language. Also, it relies more on the quality of the specifications, because the formulation of a specification determines how easy it is to use in verification.

Writing the method specifications was a non-trivial exercise. Many subtleties, like the fact that nothing sensible can be specified if a collection contains itself, only became clear during the verification.

Translating the JML specifications by hand was a good exercise for understanding what such specifications actually mean. A problem was that often small clauses were forgotten, which required that the whole proof was redone. Automatic translation would ensure that this will not happen.

Within the verification, we noticed that there was much repetition in the proofs. Using appropriate tactics and rewrite rules could significantly shorten the proofs (and hopefully also speed up the verification). In particular, a more systematic approach for dealing with abstract variables would be desirable. Also, appropriate rewrite rules for dealing with heap_equality, stack_equality and static_equality would be helpful. During verifications we already started experimenting with this. Hopefully in the future this can be fine-tuned. Also more study is necessary on how to deal with local changes in memory, *i.e.* changes in one object which do not influence the values of another object.

Finally the Hoare logic for JAVA turned out to be very useful again. Some experiments have been done with letting ISABELLE select the appropriate proof rule, but most of the methods were not very suited for this, because they almost completely consisted of method calls. Experiments with this on methods without method calls will be interesting future work. It seems that in particular fine-tuning will be needed to deal with assignments.

Chapter 8

Concluding remarks

This thesis describes the first steps of a project aimed at formal verification of JAVA programs. The work presented here is part of a larger project called LOOP, for Logic of Object Oriented Programming.

A semantics for JAVA is described in type theory and it is shown how this semantics forms the basis for program verification. The verifications are done with the use of interactive theorem provers. Typically, program verification involves big goals, but relatively simple proofs. Often, big parts of the proof consist of rewriting only. Also, different branches of the proof are often very similar. Therefore, the use of an interactive theorem prover can be very profitable in these kind of applications: by fine-tuning the theorem prover most of the 'simple proving' can be done automatically, and a user can concentrate on the essential parts of the proof. Another benefit of using a theorem prover is that it helps in avoiding the introduction of mistakes. The tool can check that no branch is forgotten, no typing error is introduced *etc.* For the verifications presented in this thesis, two theorem provers are used: PVS and ISABELLE. Both theorem provers are described in some detail, resulting in a comparison of the strong and weak points of both systems. Below, we will discuss how these two theorem provers compare in the verifications that are actually done within the LOOP project.

The LOOP project resulted in the construction of the so-called LOOP compiler, which takes JAVA classes as input and returns PVS or ISABELLE theories as output. Thus, to reason about a particular class, one only has to run the compiler on it, and the resulting files can be loaded into the theorem prover. Together with several theories describing the basic semantics of JAVA, these files describe the semantics of the translated classes. An advantage of this approach is that an arbitrary user does not have to understand all the details of the semantic encoding: he can simply use the compiler and reason about the translated classes within a theorem prover.

This thesis also briefly describes a specification language for JAVA, called JML (JAVA modeling language). This language can be used to specify JAVA classes. Currently, the LOOP compiler is being extended to generate appropriate proof obligations for classes, based on these specifications. In this thesis, the proof obligations, *i.e.* what one actually wishes to express about a JAVA class, are still formulated by hand.

It should be emphasised that the work presented in this thesis is only the first – but essential – step in the LOOP project. The semantics that has been developed so far cover almost all of sequential JAVA, including many (messy) semantical details, such as abrupt termination, exception handling, side-effects, static initialisation (not described in this thesis) and late binding. Getting this semantics right is an intellectual exercise in itself. Two non-trivial case studies are described in this thesis, and another case study has been caried out recently [BJP00]. An

important factor in all these verifications has been to find the appropriate way of expressing and proving properties. This resulted in the Hoare logic for JAVA, as presented in Chapter 5. The use of this Hoare logic made reasoning about loops easier, but still not perfect. Therefore, currently the Hoare logic is adapted to allow different output options in the postcondition [JP00a]. It is important to realise that the verification method that is used in this thesis is still under development, and with every case study it is improved.

The case studies in this thesis were time-consuming and one may wonder whether it is really worth spending so much time on such relatively simple verifications, but it is important to realise that (1) it is one of the first times that such big verifications have been done at all, and (2) the experience gained in these verifications are necessary to make the verification process easier and faster. It is our hope that in the future, it will pay off to write formal specifications and verify these specifications for widely used, general library classes. Although this will probably not be established in the near future, current work, including this thesis, shows that eventually it will be a reachable goal.

8.1 Current and future work in the LOOP project

Current work in the LOOP project focuses on the following aspects.

- Verification of JAVA card programs. To program smart cards, a restricted subset of the JAVA programming language is available (without for example multi-dimensional arrays and concurrency). Smart card programs are typically smaller then traditional JAVA programs. There is limited memory on a smart card, therefore the virtual machine on the JAVA card is smaller than the standard virtual machine and leaves out some security checks. The combination of these factors makes verification of JAVA card programs an ideal research topic for the LOOP project. It is easier to reason about these programs, and at the same time, there is much interest in their formal verification. Currently, work in the LOOP project focuses on specification and verification of the JAVA card API [PBJ00, BJP00].

- Generating proof obligations from a JAVA program and its specification. To write specifications of JAVA programs, a language called JML has been developed [LBR98] (presented in Chapter 6). Currently, the LOOP tool is being extended to translate a JAVA class with JML annotations into a series of PVS or ISABELLE theories which contain both a semantic description and proof obligations for the translated class. Therefore, a semantics for JML is under development [BPJ00]. In this thesis the language JML is already used to write specifications, but the translation to proof obligations is still done by hand.

Interesting future work would be to look at possible combinations with the Extended Static Checker (ESC) [DLNS98]. This tool performs automatically static checks on JAVA programs, preventing for example `NullPointerException`'s and `ArrayIndexOutOfBoundsException`'s. To use ESC on a JAVA program, this program should be annotated with pre- and postconditions, modifies clauses *etc*. The annotation input language for ESC is a subset of JML. A natural way to combine ESC and the LOOP compiler would be to annotate a JAVA program, check it with the static checker and finally verify the crucial parts using PVS or ISABELLE. The static checker then works as a kind of preprocessor, already finding the "easy" bugs in the program.

Also, it will be interesting to look at possibilities for combinations with other tools or formal methods. Model checkers can be used to verify automatically particular properties of JAVA programs. Abstraction techniques probably can be used to extract the crucial steps from a program. Verification of these properties can then be done on this abstract level (provided that the abstraction function and its inverse preserve the correctness of the property).

8.2 A comparison of PVS and Isabelle (part II)

This thesis concentrates in particular on the use of theorem provers in the verification of JAVA classes. Within the project, two theorem provers are used: PVS and ISABELLE. Both have been applied in case studies to reasonably large verifications. One of the reasons to use two theorem provers as output targets of the LOOP compiler is that we are interested in comparing the proof efforts in the two tools. As we want to have a high degree of automation in the proving process, we restricted our attention to theorem provers with powerful proving strategies. Chapter 3 presents a general comparison of PVS and ISABELLE, here we discuss some more application specific differences, based on our experiences in verifying JAVA programs.

Thus far, in our proofs we have mainly used rewriting to achieve automation. Both PVS and ISABELLE are good at rewriting, but there are some notable differences. As already explained in Chapter 3 (Section 3.3.3), rewriting in ISABELLE is eager, while rewriting in PVS is lazy. For our semantics of classes with static initialisations eager rewriting can cause problems, as it might not terminate. To prevent that the ISABELLE simplifier loops on these examples, the definitions dealing with static initialisation have to be unfolded explicitly, before rewriting.

However, there are also several cases where rewriting in ISABELLE is more effective, because ISABELLE is able to decide how rewrite rules should be applied, based on the assumptions in the subgoal. This is best illustrated with an example. It is easy to prove the following lemma heapmem_getbyte (and many similar ones) about the operations on the object memory.

– TYPE THEORY

$$\begin{array}{l}\forall x, y : \mathsf{OM}.\ \forall m : \mathsf{MemLoc}.\ \forall c : \mathsf{CellLoc}. \\ \quad \mathsf{heap\_equality}(x, y) \ \wedge\ m < \mathsf{heaptop}\, x \supset \\ \quad\quad \mathsf{get\_byte}\ (\mathsf{heap}(\mathsf{ml} = m, \mathsf{cl} = c))\, y = \\ \quad\quad\quad \mathsf{get\_byte}\ (\mathsf{heap}(\mathsf{ml} = m, \mathsf{cl} = c))\, x\end{array}$$

When reasoning with specifications (like in the collection case study in Section 7.2), this kind of lemmas are very useful for rewriting. In ISABELLE this works very well: if the lemma heapmem_getbyte is added to the simplifier and the subgoal contains an assumption $\mathsf{heap\_equality}(x, y)$, then every occurrence of $\mathsf{get\_byte}\ (\mathsf{heap}(\mathsf{ml} = m, \mathsf{cl} = c))\, y$ is rewritten into $\mathsf{get\_byte}\ (\mathsf{heap}(\mathsf{ml} = m, \mathsf{cl} = c))\, x$. In contrast, adding the lemma heapmem_getbyte to the PVS rewriter does not have this effect. The difference is that ISABELLE also tries matching the conditions of a rewrite rule to decide how an expression can be rewritten, while PVS does not. Therefore, PVS does not know which instantiation to choose for the variable x, and thus does not apply the rewrite rule. If we want to use this kind of rewriting in PVS, we have to give rewrite rules like the following lemma, where norm? is a recogniser function, such that $\mathsf{norm?}(x)$ is true iff x is tagged with norm.

– TYPE THEORY

$$\forall x : \mathsf{OM}.\ \forall s : \mathsf{OM} \to \mathsf{StatResult[OM]}.\ \forall m : \mathsf{MemLoc}.\ \forall c : \mathsf{CellLoc}.$$
$$\mathsf{norm?}(s\,x) \wedge \mathsf{heap\_equality}(x, (s\,x).\mathsf{ns}) \wedge m < \mathsf{heaptop}\,x \supset$$
$$\mathsf{get\_byte}\ (\mathsf{heap}(\mathsf{ml} = m, \mathsf{cl} = c))\ ((s\,x).\mathsf{ns}) =$$
$$\mathsf{get\_byte}\ (\mathsf{heap}(\mathsf{ml} = m, \mathsf{cl} = c))\,x$$

Using this rule, PVS knows exactly how to rewrite expressions matching the left hand side (provided the conditions are satisfied). This rule applies for normal terminating statements only. To use this kind of rewriting effectively in all cases, similar rules should be given for all possible kinds of termination, both for statements and expressions. This would thus result in a substantial number of rewrite rules.

Similar kind of rewrite lemmas can be generated for model variables as well (depending on their represents clauses). In ISABELLE this would only require a single rule per model variable, in PVS there would be seven. Loading all these rules in the simplifier will not improve the memory usage (and therewith the speed) of PVS.

Another feature of ISABELLE which can improve the automation of the proof process is its proof techniques based on resolution. This is used in combination with the Hoare logic presented in Chapter 5. As explained in Chapter 3, resolution tries to unify a conclusion of a theorem with the conclusion of a goal. If this succeeds it replaces the conclusion of the goal with the assumptions of the theorem. Variables in the assumptions that do not occur in the conclusions become meta-variables which can be unified later. Hoare logic proofs typically are constructed in this way: the correctness of a statement is shown by showing the correctness of its components. By using tactics which repeatedly try to do resolution with a set of given Hoare logic proof rules, partial and total correctness sentences can easily be decomposed in smaller components.

As a very small example, consider the following JAVA method (where `a` and `b` are declared as `int` in the class containing the method `m`).

– JAVA

```
void m () {
  a = 3;
  b = a;
}
```

For this method body, the following property can be proven.

– TYPE THEORY

$$[\mathsf{true}]\ \mathsf{m}\ [\lambda x : \mathsf{a}\,x = 3 \wedge \mathsf{b}\,x = 3.]$$

Of course, this property is trivial to show by automatic rewriting after unfolding the definition of TotalNormal?. However, in ISABELLE it is not even necessary to unfold this definition. By giving an appropriate set of Hoare logic rules to the systems and allowing simplification in the assertions (to simplify the substitution in the precondition) this property can be proven automatically. Important for this approach is that the assertions in the conclusions of the Hoare logic rules should contain as little structure as possible, so that they can easily be unified with the conclusion of the subgoal.

This is in particular useful when reasoning about larger methods, possibly containing loops. Ideally, the system decomposes the whole method body until only the correctness of the loop body remains to be shown (which, after instantiation of the invariant and variant can be done by the same tactic again). In the collection case study (Section 7.2) already some experiments have been done with this approach. However, because of the use of abstract methods, still much user interaction was required, because the pre- and postconditions of the method specifications that were used could not easily be unified. Future work could focus on improving and fine-tuning this approach.

The obvious question thus arises whether PVS or ISABELLE is better for the verification work in the LOOP project. Unfortunately, it is impossible to give an absolute answer to this question, as both systems have their weak and strong points which influenced our verifications.

First of all, in both systems we experienced serious performance problems. Improving our verification methods helped in reducing these problems, but nevertheless this remains a serious problem.

ISABELLE provides the flexibility to write powerful tactics, tailored to the LOOP project approach to reasoning about JAVA programs. Much fine-tuning will be required to optimise these tactics, but we feel that this will pay off as it will make reasoning about JAVA programs easier in the end.

However, there are also some practical aspects of theorem proving where our experiences with PVS are much better. When doing a large verification, it often happens in the middle of a proof construction that one suddenly notices that an assumption is forgotten, an extra lemma is needed or something the like. In that case PVS gives the user the possibility to add this lemma to the specifications files (and prove it later) or postpone the goal that can not be proven (yet). Thus, the user has the possibility to construct the rest of the proof first and worry later about the open subgoal(s). In this way it is possible to find all the problems in the proofs at one time, solve all these problems and then rerun the proof again. In ISABELLE, in theory it is possible to do the same thing, but in practice this does not work. When reasoning about JAVA programs, the goals often become so large that they do not fit on one screen anymore. Working on the second goal in the list means that the user has to scroll to see his current goal. Some more support on these matters could make working with ISABELLE much more pleasant.

8.3 To conclude

To conclude, no matter whether one aims for PVS's `QED` or ISABELLE's `No SubGoals!`, the main point of this thesis is that using the JAVA semantics as described, and using powerful translation and reasoning tools, such as PVS and ISABELLE, it has become feasible to verify non-trivial properties of real JAVA programs.

Bibliography

[AB96] A. Ayari and D. A. Basin. Generic system support for deductive program development. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, number 1055 in LNCS, pages 313–328, 1996.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[ACH76] E.A. Ashcroft, M. Clint, and C.A.R. Hoare. Remarks on "Program proving: jumps and functions by M. Clint and C.A.R. Hoare". *Acta Informatica*, 6:317–318, 1976.

[AG95] S. Agerholm and M.J.C. Gordon. Experiments with ZF set theory in HOL and Isabelle. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications, 8th International Workshop*, number 971 in LNCS, pages 32–45. Springer-Verlag, 1995.

[AG97] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 2nd edition, 1997.

[AL97] M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *Theory and Practice of Software Development (TAPSOFT '97)*, number 1214 in LNCS, pages 682–696. Springer-Verlag, 1997.

[Ame90] P. America. Designing an object-oriented programming language with behavioural subtyping. In J.W. de Bakker, W.P. de Roever, and G.Rozenberg, editors, *Foundations of Object-Oriented Languages*, number 489 in LNCS, pages 60–90. Springer-Verlag, 1990.

[AO97] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 2nd rev. edition, 1997.

[Apt81] K.R. Apt. Ten years of Hoare's logic: A survey–part I. *ACM Trans. on Progr. Lang. and Systems*, 3(4):431–483, 1981.

[Asp00] D. Aspinall. Proof General: A generic tool for proof development. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, number 1785 in LNCS, pages 38–42. Springer-Verlag, 2000.

[Bak80] J.W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice Hall, 1980.

[Bar96] H. Barendregt. The quest for correctness. In *Images of SMC research 1996*, pages 39–58. Stichting Mathematisch Centrum, 1996.

[BBC+99] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhére, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq Proof Assistant reference manual version 6.3.1, 1999.

[BCP97] K.B. Bruce, L. Cardelli, and B.C. Pierce. Comparing object encodings. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software*, number 1281 in LNCS, pages 415–438. Springer-Verlag, 1997.

[BDJ+00] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, S. de Sousa, and S. Yu. Formalization in Coq of the Java Card virtual machine. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G.T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, number 269 - 5/2000 in Informatik Berichte FernUniversität Hagen, pages 50–56, 2000.

[BHJP00] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P.D. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, number 1827 in LNCS, pages 1–21. Springer-Verlag, 2000.

[BJP00] J. van den Berg, B. Jacobs, and E. Poll. Formal Specification and Verification of JavaCard's Application Identifier Class. In I. Attali and Th. Jensen, editors, *Proceedings of the JavaCard Workshop*, 2000. INRIA Proceedings.

[BK91] D. Basin and M. Kaufmann. The Boyer-Moore prover and Nuprl: An experimental comparison. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 90–119. Cambridge University Press, 1991.

[BL99] J. Bergstra and M. Loots. Empirical semantics for object-oriented programs. Artificial Intelligence Preprint Series nr. 007, Department of Philosophy, Utrecht University, 1999.

[Boe99] F.S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Foundations of Software Science and Computation Structures*, number 1578 in LNCS, pages 135–149. Springer-Verlag, 1999.

[BPJ00] J. van den Berg, E. Poll, and B. Jacobs. First steps in formalising JML. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G.T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, number 269 - 5/2000 in Informatik Berichte FernUniversität Hagen, pages 103–110, 2000.

[Bru70] N.G de Bruijn. The mathematical language AUTOMATH. Number 25 in Lect. Notes Math., pages 29–61. Springer-Verlag, 1970.

[BS99] E. Börger and W. Schulte. Programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in LNCS, pages 353–404. Springer-Verlag, 1999.

[Bud00] T. Budd. *Understanding Object-oriented programming with Java – updated edition.* Addison-Wesley, 2000.

[CAB+86] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Inf & Comp.*, 76(2/3):138–164, 1988.

[CD96] J. Crow and B.L. Di Vito. Formalizing Space Shuttle software requirements. In *First Workshop on Formal Methods in Software Practice (FMSP '96)*, pages 40–48. ACM, 1996.

[CGJ99] S. Coupet-Grimal and L. Jakubiec. Hardware verification using co-induction in COQ. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference (TPHOLs '99)*, number 1690 in LNCS, pages 91–108. Springer-Verlag, 1999.

[CH72] M. Clint and C.A.R. Hoare. Program proving: jumps and functions. *Acta Informatica*, 1:214–224, 1972.

[Chr84] F. Christian. Correct and robust programs. *IEEE Trans. on Software Eng.*, 10(2):163–174, 1984.

[CLK98] P. Chan, R. Lee, and D. Kramer. *The Java Class Libraries, Second Edition, Volume 1.* Addison-Wesley, 2nd edition, 1998.

[CM95] V.A. Carreño and P.S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *Higher Order Logic Theorem Proving and Its Applications, 8th International Workshop*, 1995. Category B proceedings, available at `http://lal.cs.byu.edu/lal/hol95/Bprocs/indexB.html`.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, 1995. Available, with specification files, at `http://www.csl.sri.com/wift-tutorial.html`.

[CP95] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Inf & Comp.*, 114(2):329–350, 1995.

[DAR] Database of existing mechanized reasoning systems. `http://www-formal.stanford.edu/clt/ARS/systems.html`.

[DL96] K.K. Dhara and G.T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings 18th International Conference on Software Engineering*, pages 258–267. IEEE, 1996.

[DLN98] D.L. Detlefs, K.R.M. Leino, and G. Nelson. Wrestling with rep exposure. Technical Report SRC 156, Digital System Research Center, 1998.

[DLNS98] D.L. Detlefs, K.R.M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. Technical Report SRC 159, Digital System Research Center, 1998.

[DMN70] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Simula 67 common base language. Technical Report N.S-22, Norwegian Computing Center, 1970.

[Eng98] J. English. The story of the Java platform, 1998. `http://java.sun.com/nav/whatis/storyofjava.html`.

[Fok78] M.M. Fokkinga. Axiomatization of declarations and the formal treatment of an escape construct. In E.J. Neuhold, editor, *Formal Descriptions of Programming Language Concepts*, pages 221–235. IFIP TC-2 (Working Group 2.2), North-Holland, 1978.

[GH98] W.O.D. Griffioen and M. Huisman. A comparison of PVS and Isabelle/HOL. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference (TPHOLs '98)*, number 1479 in LNCS, pages 123–142, 1998.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, 2000.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL, A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[GMW79] M.J.C. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Number 78 in LNCS. Springer-Verlag, 1979.

[Gor88] M.J.C. Gordon. *Programming Language Theory and its Implementation*. Prentice Hall, 1988.

[Gor89] M.J.C. Gordon. Mechanizing programming logics in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

[Gor95] M.J.C. Gordon. Notes on PVS from a HOL perspective. Available at `http://www.cl.cam.ac.uk/users/~mjcg/PVS.html`, 1995.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[Gri00] W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. PhD thesis, Computing Science Institute, University of Nijmegen, 2000.

[Har98] J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[HBL99] P.H. Hartel, M.J. Butler, and M. Levy. The operational semantics of a Java Secure Processor. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in LNCS, pages 313–352. Springer, 1999.

[HHJT98] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In C. Hankin, editor, *Proceedings of European Symposium on Programming (ESOP '98)*, number 1381 in LNCS, pages 105–121. Springer-Verlag, 1998.

[HJ98] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Inf & Comp.*, 145:107–152, 1998.

[HJ00a] M. Huisman and B. Jacobs. Inheritance in higher order logic: Modeling and reasoning. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference (TPHOLs 2000)*, number 1869 in LNCS, pages 301–319. Springer-Verlag, 2000.

[HJ00b] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE 2000)*, number 1783 in LNCS, pages 284–303. Springer-Verlag, 2000.

[HJB00] M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. *Software Tools for Technology Transfer (STTT)*, 2000. To appear.

[HNSS98] M. Hofmann, W. Naraschewski, M. Steffen, and T. Stroup. Inheritance of proofs. *Theory & Practice of Object Systems*, 4(1):51–69, 1998.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Jac96] B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conference on Object-Oriented Programming*, number 1098 in LNCS, pages 210–231. Springer-Verlag, 1996.

[Jac00] B. Jacobs. A formalisation of Java's exception mechanism. Technical Report CSI-R0015, Computing Science Institute, University of Nijmegen, 2000.

[Jav] JavaTM 2 platform, standard edition, version 1.3 API specification. `http://www.java.sun.com/j2se/1.3/docs/api/index.html`.

[JBH+98] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '98)*, pages 329–340. ACM Press, 1998.

[JP00a] B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. Technical Report CSI-R0018, Computing Science Institute, University of Nijmegen, 2000.

[JP00b] B. Jacobs and E. Poll. A monad for basic Java semantics. In T. Rus, editor, *Algebraic Methodology and Software Technology (AMAST 2000)*, number 1816 in LNCS, pages 150–164. Springer, Berlin, 2000.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[Kro99] T. Kropf. Recent advancements in hardware verification - how to make theorem proving fit for an industrial usage. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference (TPHOLs '99)*, number 1690 in LNCS, pages 1–4. Springer-Verlag, 1999.

[KWP99] F. Kammüller, M. Wenzel, and L.C. Paulson. Locales. a sectioning concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference (TPHOLs '99)*, number 1690 in LNCS, pages 149–165. Springer-Verlag, 1999.

[LBR98] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06, Iowa State University, Department of Computer Science, 1998.

[LBR99] G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and W. Harvey, editors, *Behavioral Specifications for Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[LD00] G.T. Leavens and K.K. Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In G.T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 113–135. Cambridge University Press, 2000.

[Lea93] G.T. Leavens. Inheritance of interface specifications (extended abstract). Technical Report 93-23, Iowa State University, Department of Computer Science, 1993. Appears in the Workshop on Interface Definition Languages, WIDL '94.

[Lei95] K.R.M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Inst. of Techn., 1995.

[Lei98] K.R.M. Leino. Data groups: specifying the modification of extended state. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '98)*, pages 144–153. ACM Press, 1998.

[LP80] D.C. Luckham and W. Polak. Ada exception handling: an axiomatic approach. *ACM Trans. on Progr. Lang. and Systems*, 2:225–233, 1980.

[LP92] Z. Luo and R. Pollack. *LEGO Proof Development System: User's Manual*. Department of Computer Science, University of Edinburgh, 1992.

[LS90] K. Lodaya and R.K. Shyamasundar. Proof theory for exception handling in a tasking environment. *Acta Informatica*, 28:7–41, 1990.

[LS97] K.R.M. Leino and R. Stata. Checking object invariants. Technical Report SRC 1997-007, Digital System Research Center, 1997.

[LvdS94] K.R.M. Leino and J. van de Snepscheut. Semantics of exceptions. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, pages 447–466. North-Holland, 1994.

[LW94] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Trans. on Progr. Lang. and Systems*, 16(1):1811–1841, 1994.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd rev. edition, 1997.

[MH96] N.A. Merriam and M.D. Harrison. Evaluating the interfaces of three theorem proving assistants. In F. Bodart and J. Vanderdonckt, editors, *Proceedings of the 3rd International Eurographics Workshop on Design, Specification, and Verification of Interactive Systems*, Eurographics Series. Springer-Verlag, 1996.

[Mit90] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Principles of Programming Languages*, pages 109–124. ACM Press, 1990.

[ML82] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North Holland, Amsterdam, 1982.

[MPH97] P. Müller and A. Poetzsch-Heffter. Formal specification techniques for object-oriented programs. In M. Jarke, K. Pasedach, and K. Pohl, editors, *Informatik 97: Informatik als Innovationsmotor*, Informatik Aktuell. Springer-Verlag, 1997.

[MPH00a] J. Meyer and A. Poetzsch-Heffter. An architecture of interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 63–77. Springer-Verlag, 2000.

[MPH00b] P. Müller and A. Poetzsch-Heffter. Modular specification and verfication techniques for object-oriented software components. In G.T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 137–159. Cambridge University Press, 2000.

[Nor98] M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, Computer Lab, 1998. Available as Technical Report No. 453.

[Nor99] M. Norrish. Deterministic expressions in C. In S.D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, number 1576 in LNCS, pages 147–161. Springer-Verlag, 1999.

[NW98] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, number 1479 in LNCS, pages 349–366. Springer-Verlag, 1998.

[Ohe00] D. von Oheimb. Axiomatic semantics for Java^{light}. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G.T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, number 269 - 5/2000 in Informatik Berichte FernUniversität Hagen, pages 88–95, 2000.

[ON99] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in LNCS, pages 119–156. Springer, 1999.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification (CAV '96)*, number 1102 in LNCS, pages 411–414. Springer-Verlag, 1996.

[OSRSC99a] S. Owre, N. Shankar, J.M. Rushby, and D. Stringer-Calvert. PVS language reference, 1999. Version 2.3.

[OSRSC99b] S. Owre, N. Shankar, J.M. Rushby, and D. Stringer-Calvert. PVS system guide, 1999. Version 2.3.

[Par83] D. Parnas. A generalized control structure and its formal definition. *Communications of the ACM*, 26(8):572–581, 1983.

[Pau90] L.C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pau94] L.C. Paulson. *Isabelle - a generic theorem prover*. Number 828 in LNCS. Springer-Verlag, 1994. With contributions by Tobias Nipkow.

[PBJ00] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In *Fourth Smart Card Research and Advanced Application Conference (IFIP Cardis 2000)*. Kluwer Academic Publishers, 2000.

[PD98] F. Puitg and J.-F. Dufourd. Formal specification and theorem proving breakthroughs in geometric modeling. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference (TPHOLs '98)*, number 1479 in LNCS, pages 401–422, 1998.

[Pel86] F.J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986. Errata, JAR 4 (1988), 235–236 and JAR 18 (1997), 135.

[Pfe] F. Pfenning. Isabelle bibliography. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/biblio.html`.

[PH97] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habil. Thesis, Techn. University München, 1997.

[PHM98] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W.P. de Roever, editors, *Programming Concepts and Methods (PROCOMET)*, IFIP, pages 404–423. Chapman & Hall, 1998.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, number 1576 in LNCS, pages 162–176. Springer-Verlag, 1999.

[Pol00] E. Poll. A coalgebraic semantics of subtyping. In H. Reichel, editor, *Coalgebraic Methods in Computer Science*, number 33 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 2000.

[Pra95] W. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms*. PhD thesis, Utrecht University, 1995.

[Pus99] C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W.R. Claeveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, number 1579 in LNCS, pages 89–103. Springer-Verlag, 1999.

[PVS] PVS buglist. `http://pvs.csl.sri.com/htbin/pvs-bug-list/`.

[Qia99] Z. Qian. A formal specification of Java™ Virtual Machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in LNCS, pages 271–311. Springer, 1999.

[Rei95] H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.

[Rey98] J.C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[RJT00] J. Rothe, B. Jacobs, and H. Tews. The coalgebraic class specification language ccsl. In *4th workshop on: Tools for System Design and Verification*, 2000.

[ROS98] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[Rud92] P. Rudnicki. An overview of the MIZAR project, 1992.
Unpublished; available by anonymous FTP from `menaik.cs.ualberta.ca` as `put/Mizar/Mizar-Over.tar.Z`.

[Rus] J. Rushby. PVS bibliography.
`http://www.csl.sri.com/~rushby/pvs-bib.html`.

[Rus99] J. Rushby. Mechanized formal methods: Where next? In J.M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods (FM '99)*, number 1708 in LNCS, pages 48–51. Springer-Verlag, 1999.

[RV98] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension of ML. *Theory & Practice of Object Systems*, 4(1):27–50, 1998.

[SORSC99] N. Shankar, S. Owre, J.M. Rushby, and D. Stringer-Calvert. PVS prover guide, 1999. Version 2.3.

[Sym99] D. Syme. Proving java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in LNCS, pages 83–118. Springer, 1999.

[Tew00] H. Tews. *Coalgebraic Specification and Verification*. PhD thesis, Technical University of Dresden, 2000. Manuscript.

[Vec] `Vector` *class (copyright Sun Microsystems, version number 1.38, 1997), with JML annotations*. Loop web pages: `http://www.cs.kun.nl/~bart/LOOP/Vector_annotated.java`.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *16'th ACM Symposium on Principles of Programming Languages*, 1989.

[Wen95] M. Wenzel. Using axiomatic type classes in Isabelle, a tutorial, 1995.

[Wen97] M. Wenzel. Type classes and overloading in higher-order logic. In E.L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference (TPHOLs '97)*, number 1275 in LNCS, pages 307–322. Springer-Verlag, 1997.

[Wen99] M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference (TPHOLs '99)*, number 1690 in LNCS, pages 167–184. Springer-Verlag, 1999.

[WM95] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.

[You97] W.D. Young. Comparing verification systems: Interactive Consistency in ACL2. *IEEE Transactions on Software Engineering*, 23(4):214–223, 1997.

[Zam97] V. Zammit. A comparative study of Coq and HOL. In E.L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference (TPHOLs '97)*, number 1275 in LNCS, pages 323–338. Springer-Verlag, 1997.

Subject Index

Java Semantics Index

Definition and Symbol Index

Appendix A

Hoare logic rules

This appendix presents the rules from the Hoare logic presented in Chapter 5. We present the rules for normal correctness, exception correctness, and return correctness. The rules for break correctness and continue correctness are similar to the rules for return correctness.

All these rules have been proven sound *w.r.t* our JAVA semantics as presented in Chapter 2. The soundness proofs have been done both in PVS and in ISABELLE.

For readibility we use the following abbreviations.

$$
\begin{array}{l}
P, Q : \mathsf{OM} \to \mathsf{bool} \vdash \\
\quad P \wedge Q : \mathsf{bool} \stackrel{\text{def}}{=} \\
\quad\quad \lambda x : \mathsf{OM}.\ P\,x \wedge Q\,x
\end{array}
$$

$$
\begin{array}{l}
C : \mathsf{OM} \to \mathsf{ExprResult[OM, bool]} \vdash \\
\quad \mathsf{norm}(C) : \mathsf{bool} \stackrel{\text{def}}{=} \\
\quad\quad \lambda x : \mathsf{OM}.\ \mathsf{CASE}\ c\,x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{false} \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\,x \mapsto \mathsf{true} \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\,a \mapsto \mathsf{false}\}
\end{array}
$$

$$
\begin{array}{l}
C : \mathsf{OM} \to \mathsf{ExprResult[OM, bool]} \vdash \\
\quad \mathsf{true}(C) : \mathsf{bool} \stackrel{\text{def}}{=} \\
\quad\quad \lambda x : \mathsf{OM}.\ \mathsf{CASE}\ c\,x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{false} \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\,x \mapsto x.\mathsf{res} \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\,a \mapsto \mathsf{false}\}
\end{array}
$$

$$
\begin{array}{l}
C : \mathsf{OM} \to \mathsf{ExprResult[OM, bool]} \vdash \\
\quad \mathsf{false}(C) : \mathsf{bool} \stackrel{\text{def}}{=} \\
\quad\quad \lambda x : \mathsf{OM}.\ \mathsf{CASE}\ c\,x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{false} \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\,x \mapsto \neg x.\mathsf{res} \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\,a \mapsto \mathsf{false}\}
\end{array}
$$

A.1 Normal correctness of statements

$$\begin{array}{l}\text{pre, post}: \text{Self} \to \text{bool}, \text{stat}: \text{Self} \to \text{StatResult[Self]} \vdash \\ \quad \text{PartialNormal?(pre, stat, post)} : \text{bool} \stackrel{\text{def}}{=} \\ \quad\quad \forall x : \text{Self}.\, \text{pre}\, x \supset \text{CASE stat}\, x \text{ OF } \{ \\ \quad\quad\quad\quad \mid \text{hang} \mapsto \text{true} \\ \quad\quad\quad\quad \mid \text{norm}\, y \mapsto \text{post}\, y \\ \quad\quad\quad\quad \mid \text{abnorm}\, a \mapsto \text{true} \}\end{array}$$

$$\begin{array}{l}\text{pre, post}: \text{Self} \to \text{bool}, \text{stat}: \text{Self} \to \text{StatResult[Self]} \vdash \\ \quad \text{TotalNormal?(pre, stat, post)} : \text{bool} \stackrel{\text{def}}{=} \\ \quad\quad \forall x : \text{Self}.\, \text{pre}\, x \supset \text{CASE stat}\, x \text{ OF } \{ \\ \quad\quad\quad\quad \mid \text{hang} \mapsto \text{false} \\ \quad\quad\quad\quad \mid \text{norm}\, y \mapsto \text{post}\, y \\ \quad\quad\quad\quad \mid \text{abnorm}\, a \mapsto \text{false} \}\end{array}$$

Notation:

$$\text{PartialNormal?}(P, S, Q) \stackrel{\text{def}}{=} \{P\}\, S\, \{Q\}$$

$$\text{TotalNormal?}(P, S, Q) \stackrel{\text{def}}{=} [P]\, S\, [Q]$$

Partial skip rule:

$$\{P\}\ \text{skip}\ \{P\}$$

Total skip rule:

$$[P]\ \text{skip}\ [P]$$

Partial precondition strengthening:

$$\frac{\forall x : \text{OM}.\, P\, x \supset R\, x \qquad \{R\}\, S\, \{Q\}}{\{P\}\, S\, \{Q\}}$$

Total precondition strengthening:

$$\frac{\forall x : \text{OM}.\, P\, x \supset R\, x \qquad [R]\, S\, [Q]}{[P]\, S\, [Q]}$$

Partial postcondition weakening:

$$\frac{\forall x : \text{OM}.\, R\, x \supset Q\, x \qquad \{P\}\, S\, \{R\}}{\{P\}\, S\, \{Q\}}$$

Total postcondition weakening:

$$\frac{\forall x : \text{OM}.\, R\, x \supset Q\, x \qquad [P]\, S\, [R]}{[P]\, S\, [Q]}$$

Partial composition rule:

$$\frac{\{P\}\, S\, \{R\} \qquad \{R\}\, T\, \{Q\}}{\{P\}\, S\,;\, T\, \{Q\}}$$

Total composition rule:

$$\frac{[P]\, S\, [R] \qquad [R]\, T\, [Q]}{[P]\, S\,;\, T\, [Q]}$$

Partial deep composition rule:

$$\frac{\{P\}\, S\, \{\lambda x : \mathsf{OM}.\ Q\ (f\ x)\}}{\{P\}\, S\, @@\, f\, \{Q\}}$$

Total deep composition rule:

$$\frac{[P]\, S\, [\lambda x : \mathsf{OM}.\ Q\ (f\ x)]}{[P]\, S\, @@\, f\, [Q]}$$

Partial stacktop_inc rule:

$$\{\lambda x : \mathsf{OM}.\ P\ ((\mathsf{stacktop\_inc}\, x).\mathsf{ns})\}\ \mathsf{stacktop\_inc}\ \{P\}$$

Total stacktop_inc rule:

$$[\lambda x : \mathsf{OM}.\ P\ ((\mathsf{stacktop\_inc}\, x).\mathsf{ns})]\ \mathsf{stacktop\_inc}\ [P]$$

Partial stacktop_inc rule empty stack:

$$\frac{\forall x : \mathsf{OM}.\ \forall t : \mathsf{MemLoc}.\ \mathsf{stacktop}\ x \leq t \supset \mathsf{stackmem}\ x\ t = \mathsf{EmptyObjectCell}}{\{P\}\ \mathsf{stacktop\_inc}\ \{\lambda x : \mathsf{OM}.\ P\ (\mathsf{stacktop\_dec}\, x)\}}$$

Total stacktop_inc rule empty stack:

$$\frac{\forall x : \mathsf{OM}.\ \forall t : \mathsf{MemLoc}.\ \mathsf{stacktop}\ x \leq t \supset \mathsf{stackmem}\ x\ t = \mathsf{EmptyObjectCell}}{[P]\ \mathsf{stacktop\_inc}\ [\lambda x : \mathsf{OM}.\ P\ (\mathsf{stacktop\_dec}\, x)]}$$

Partial IF-THEN rule:

$$\frac{\{P\ \wedge\ \mathsf{true}(C)\}\ \mathsf{E2S}(C)\,;\, S\, \{Q\} \qquad \{P\ \wedge\ \mathsf{false}(C)\}\ \mathsf{E2S}(C)\, \{Q\}}{\{P\}\ \mathsf{IF\text{-}THEN}(C)(S)\, \{Q\}}$$

Total IF-THEN rule:

$$\frac{[P\ \wedge\ \mathsf{true}(C)]\ \mathsf{E2S}(C)\,;\, S\, [Q] \qquad [P\ \wedge\ \mathsf{false}(C)]\ \mathsf{E2S}(C)\, [Q]}{[P\ \wedge\ \mathsf{norm}(C)]\ \mathsf{IF\text{-}THEN}(C)(S)\, [Q]}$$

Partial IF-THEN-ELSE rule:

$$\frac{\{P \ \wedge \ \mathsf{true}(C)\}\ \mathsf{E2S}(C)\ ;\ S\ \{Q\} \qquad \{P \ \wedge \ \mathsf{false}(C)\}\ \mathsf{E2S}(C)\ ;\ T\ \{Q\}}{\{P\}\ \mathsf{IF\text{-}THEN\text{-}ELSE}(C)(S)(T)\ \{Q\}}$$

Total IF-THEN-ELSE rule:

$$\frac{[P \ \wedge \ \mathsf{true}(C)]\ \mathsf{E2S}(C)\ ;\ S\ [Q] \qquad [P \ \wedge \ \mathsf{false}(C)]\ \mathsf{E2S}(C)\ ;\ T\ [Q]}{[P \ \wedge \ \mathsf{norm}(C)]\ \mathsf{IF\text{-}THEN\text{-}ELSE}(C)(S)(T)\ [Q]}$$

Total CATCH-BREAK normal rule:

$$\frac{[P]\ S\ [Q]}{[P]\ \mathsf{CATCH\text{-}BREAK}(ll)(S)\ [Q]}$$

Total CATCH-CONTINUE normal rule:

$$\frac{[P]\ S\ [Q]}{[P]\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ [Q]}$$

Total CATCH-STAT-RETURN normal rule:

$$\frac{[P]\ S\ [Q]}{[P]\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(S)\ [Q]}$$

Partial WHILE rule:

$$\frac{\begin{array}{c}\{P \ \wedge \ \mathsf{true}(C)\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ \{P\} \\ \{P \ \wedge \ \mathsf{false}(C)\}\ \mathsf{E2S}(C)\ \{Q\}\end{array}}{\{P\}\ \mathsf{WHILE}(ll)(C)(S)\ \{Q\}}$$

Total WHILE rule:

$$\frac{\begin{array}{c}\mathsf{well\_founded?}(R) \\ \forall a.\ [P \ \wedge \ \mathsf{true}(C) \ \wedge \ \mathit{variant} = a] \\ \quad \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S) \\ \quad [P \ \wedge \ \mathsf{norm}(C) \ \wedge \ (\mathit{variant}, a) \in R] \\ \quad \{P \ \wedge \ \mathsf{false}(C)\}\ \mathsf{E2S}(C)\ \{Q\}\end{array}}{[P \ \wedge \ \mathsf{norm}(C)]\ \mathsf{WHILE}(ll)(C)(S)\ [Q]}$$

Partial FOR rule:

$$\frac{\begin{array}{c}\{P \ \wedge \ \mathsf{true}(C)\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ ;\ U\ \{P\} \\ \{P \ \wedge \ \mathsf{false}(C)\}\ \mathsf{E2S}(C)\ \{Q\}\end{array}}{\{P\}\ \mathsf{FOR}(ll)(C)(U)(S)\ \{Q\}}$$

Total FOR rule:

$$\frac{\begin{array}{c}\mathsf{well\_founded?}(R)\\ \forall a.\ [P\ \wedge\ \mathsf{true}(C)\ \wedge\ \mathit{variant} = a]\\ \quad \mathsf{E2S}(C)\,;\,\mathsf{CATCH\text{-}CONTINUE}(ll)(S)\,;\,U\\ \quad [P\ \wedge\ \mathsf{norm}(C)\ \wedge\ (\mathit{variant}, a) \in R]\\ \quad \{P\ \wedge\ \mathsf{false}(C)\}\ \mathsf{E2S}(C)\ \{Q\}\end{array}}{[P\ \wedge\ \mathsf{norm}(C)]\ \mathsf{FOR}(ll)(C)(U)(S)\ [Q]}$$

Partial block rule:

$$\frac{\begin{array}{l}\forall y\colon \mathsf{Self} \to \mathsf{Out}.\ \forall \mathit{y\_becomes}\colon \mathsf{Self} \to \mathsf{Out} \to \mathsf{Self}.\\ \quad \{\lambda x\colon \mathsf{Self}.\ \ P\,x\ \wedge\\ \qquad\qquad y = \mathsf{get\_typ}(\mathsf{stack}(\,\mathsf{ml} = \mathsf{stacktop}\,x, \mathsf{cl} = \mathit{cl}\,))\ \wedge\\ \qquad\qquad \mathit{y\_becomes} = \mathsf{put\_typ}(\mathsf{stack}(\,\mathsf{ml} = \mathsf{stacktop}\,x, \mathsf{cl} = \mathit{cl}\,))\ \wedge\\ \qquad\qquad y\,x = \omega\}\\ \quad S(y, \mathit{y\_becomes})\\ \quad \{Q\}\end{array}}{\begin{array}{l}\{P\}\\ \ \mathsf{LET}\ y = \mathsf{get\_typ}(\mathsf{stack}(\,\mathsf{ml} = \mathit{ml}, \mathsf{cl} = \mathit{cl}\,)),\\ \qquad \mathit{y\_becomes} = \mathsf{put\_typ}(\mathsf{stack}(\,\mathsf{ml} = \mathit{ml}, \mathsf{cl} = \mathit{cl}\,))\\ \ \mathsf{IN}\ S(y, \mathit{y\_becomes})\\ \{Q\}\end{array}}$$

Total block rule:

$$\frac{\begin{array}{l}\forall y\colon \mathsf{Self} \to \mathsf{Out}.\ \forall \mathit{y\_becomes}\colon \mathsf{Self} \to \mathsf{Out} \to \mathsf{Self}.\\ \quad [\lambda x\colon \mathsf{Self}.\ \ P\,x\ \wedge\\ \qquad\qquad y = \mathsf{get\_typ}(\mathsf{stack}(\,\mathsf{ml} = \mathsf{stacktop}\,x, \mathsf{cl} = \mathit{cl}\,))\ \wedge\\ \qquad\qquad \mathit{y\_becomes} = \mathsf{put\_typ}(\mathsf{stack}(\,\mathsf{ml} = \mathsf{stacktop}\,x, \mathsf{cl} = \mathit{cl}\,))\ \wedge\\ \qquad\qquad y\,x = \omega]\\ \quad S(y, \mathit{y\_becomes})\\ \quad [Q]\end{array}}{\begin{array}{l}[P]\\ \ \mathsf{LET}\ y = \mathsf{get\_typ}(\mathsf{stack}(\,\mathsf{ml} = \mathit{ml}, \mathsf{cl} = \mathit{cl}\,)),\\ \qquad \mathit{y\_becomes} = \mathsf{put\_typ}(\mathsf{stack}(\,\mathsf{ml} = \mathit{ml}, \mathsf{cl} = \mathit{cl}\,))\\ \ \mathsf{IN}\ S(y, \mathit{y\_becomes})\\ [Q]\end{array}}$$

Partial CS2S rule:

$$\frac{\begin{array}{l}\exists \mathit{refpos}\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \exists \mathit{name}\colon \mathsf{OM} \to \mathsf{string}.\ \forall z\colon \mathsf{OM}.\\ \quad \{P\}\\ \quad \mathit{ref\_expr}\\ \quad \{\lambda x\colon \mathsf{OM}.\ \lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge\\ \qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{\\ \qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{true}\\ \qquad\qquad\quad \mid \mathsf{ref}\, r \mapsto\ \ r = \mathit{refpos}\, x \wedge\\ \qquad\qquad\qquad\qquad \mathsf{get\_type}\, r\ x = \mathit{name}\, x\}\}\\ \quad \{\lambda x\colon \mathsf{OM}.\ R\,x\ \wedge\ x = z\}\ \mathit{statement}(\mathit{coalg}(\mathit{name}\, z)(\mathit{refpos}\, z))\ \{Q\}\end{array}}{\{P\}\ \mathsf{CS2S}(\mathit{coalg})(\mathit{ref\_expr})(\mathit{statement})\ \{Q\}}$$

Total CS2S rule:

$$\frac{\begin{array}{l}\exists \mathit{refpos}\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \exists \mathit{name}\colon \mathsf{OM} \to \mathsf{string}.\ \forall z\colon \mathsf{OM}.\\ \quad [P]\\ \quad \mathit{ref\_expr}\\ \quad [\lambda x\colon \mathsf{OM}.\ \lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge\\ \qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{\\ \qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{false}\\ \qquad\qquad\quad \mid \mathsf{ref}\, r \mapsto\ \ r = \mathit{refpos}\, x \wedge\\ \qquad\qquad\qquad\qquad \mathsf{get\_type}\, r\ x = \mathit{name}\, x\}]\\ \quad [\lambda x\colon \mathsf{OM}.\ R\,x\ \wedge\ x = z]\ \mathit{statement}(\mathit{coalg}(\mathit{name}\, z)(\mathit{refpos}\, z))\ [Q]\end{array}}{[P]\ \mathsf{CS2S}(\mathit{coalg})(\mathit{ref\_expr})(\mathit{statement})\ [Q]}$$

A.2 Normal correctness of expressions

$$\begin{array}{l}\mathsf{pre}\colon \mathsf{Self} \to \mathsf{bool},\ \mathsf{post}\colon \mathsf{Self} \to \mathsf{Out} \to \mathsf{bool},\ \mathsf{expr}\colon \mathsf{Self} \to \mathsf{ExprResult[Self, Out]}\ \vdash\\ \quad \mathsf{PartialNormal?(pre, expr, post)}\ :\ \mathsf{bool} \stackrel{\mathrm{def}}{=}\\ \qquad \forall x\colon \mathsf{Self}.\ \mathsf{pre}\, x \supset\ \ \mathsf{CASE}\ \mathsf{expr}\, x\ \mathsf{OF}\ \{\\ \qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{true}\\ \qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{post}\ (y.\mathsf{ns})\ (y.\mathsf{res})\\ \qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{true}\}\end{array}$$

$$\begin{array}{l}\mathsf{pre}\colon \mathsf{Self} \to \mathsf{bool},\ \mathsf{post}\colon \mathsf{Self} \to \mathsf{Out} \to \mathsf{bool},\ \mathsf{expr}\colon \mathsf{Self} \to \mathsf{ExprResult[Self, Out]}\ \vdash\\ \quad \mathsf{TotalNormal?(pre, expr, post)}\ :\ \mathsf{bool} \stackrel{\mathrm{def}}{=}\\ \qquad \forall x\colon \mathsf{Self}.\ \mathsf{pre}\, x \supset\ \ \mathsf{CASE}\ \mathsf{expr}\, x\ \mathsf{OF}\ \{\\ \qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{false}\\ \qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{post}\ (y.\mathsf{ns})\ (y.\mathsf{res})\\ \qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{false}\}\end{array}$$

Notation:

$$\mathsf{PartialNormal?}(P, E, Q) \stackrel{\mathrm{def}}{=} \{P\}\, E\, \{\mathsf{expr}(Q)\}$$

$$\mathsf{TotalNormal?}(P, E, Q) \stackrel{\mathrm{def}}{=} [P]\, E\, [\mathsf{expr}(Q)]$$

Partial const axiom 1:

$$\{P\}\,\mathsf{const}(a)\,\{\mathsf{expr}(\lambda x : \mathsf{OM}.\,\lambda v : \mathsf{Out}.\,v = a \;\wedge\; P\,x)\}$$

Partial const axiom 2:

$$\{\lambda x : \mathsf{OM}.\,P\,x\,a\}\,\mathsf{const}(a)\,\{\mathsf{expr}(P)\}$$

Total const axiom 1:

$$[P]\,\mathsf{const}(a)\,[\mathsf{expr}(\lambda x : \mathsf{OM}.\,\lambda v : \mathsf{Out}.\,v = a \;\wedge\; P\,x)]$$

Total const axiom 2:

$$[\lambda x : \mathsf{OM}.\,P\,x\,a]\,\mathsf{const}(a)\,[\mathsf{expr}(P)]$$

Partial E2S rule:

$$\frac{\{P\}\,E\,\{\mathsf{expr}(\lambda x : \mathsf{OM}.\,\lambda v : \mathsf{Out}.\,Q\,x)\}}{\{P\}\,\mathsf{E2S}(E)\,\{Q\}}$$

Total E2S rule:

$$\frac{[P]\,E\,[\mathsf{expr}(\lambda x : \mathsf{OM}.\,\lambda v : \mathsf{Out}.\,Q\,x)]}{[P]\,\mathsf{E2S}(E)\,[Q]}$$

Partial expression precondition strengthening:

$$\frac{\forall x : \mathsf{OM}.\,P\,x \supset R\,x \qquad \{R\}\,E\,\{\mathsf{expr}(Q)\}}{\{P\}\,E\,\{\mathsf{expr}(Q)\}}$$

Total expression precondition strengthening:

$$\frac{\forall x : \mathsf{OM}.\,P\,x \supset R\,x \qquad [R]\,E\,[\mathsf{expr}(Q)]}{[P]\,E\,[\mathsf{expr}(Q)]}$$

Partial expression postcondition weakening:

$$\frac{\forall x : \mathsf{OM}.\,\forall v : \mathsf{Out}.\,R\,x\,v \supset Q\,x\,v \qquad \{P\}\,E\,\{\mathsf{expr}(R)\}}{\{P\}\,E\,\{\mathsf{expr}(Q)\}}$$

Total expression postcondition weakening:

$$\frac{\forall x : \mathsf{OM}.\,\forall v : \mathsf{Out}.\,R\,x\,v \supset Q\,x\,v \qquad [P]\,E\,[\mathsf{expr}(R)]}{[P]\,E\,[\mathsf{expr}(Q)]}$$

Partial assignment rule:

$$\frac{\{P\}\,E\,\{\mathsf{expr}(\lambda x : \mathsf{OM}.\,\lambda v : \mathsf{Out}.\,Q\;(\mathit{var\_becomes}\,x\;v)\;v)\}}{\{P\}\,\mathsf{A2E}(\mathit{var\_becomes})(E)\,\{\mathsf{expr}(Q)\}}$$

Total assignment rule:

$$\frac{[P]\,E\,[\mathsf{expr}(\lambda x:\mathsf{OM}.\,\lambda v:\mathsf{Out}.\,Q\;(\mathit{var\_becomes}\;x\;v)\;v)]}{[P]\,\mathsf{A2E}(\mathit{var\_becomes})(E)\,[\mathsf{expr}(Q)]}$$

Partial expression deep composition rule:

$$\frac{\{P\}\,E\,\{\mathsf{expr}(\lambda x:\mathsf{OM}.\,\lambda v:\mathsf{Out}.\,Q\;(f\;x)\;v)\}}{\{P\}\,S\,@@\,f\,\{\mathsf{expr}(Q)\}}$$

Total expresssion deep composition rule:

$$\frac{[P]\,E\,[\mathsf{expr}(\lambda x:\mathsf{OM}.\,\lambda v:\mathsf{Out}.\,Q\;(f\;x)\;v)]}{[P]\,S\,@@\,f\,[\mathsf{expr}(Q)]}$$

Partial binary operator $\oplus$: Out $\rightarrow$ Out $\rightarrow$ Out2 rule:

$$\frac{\begin{array}{l}\exists \mathit{expr}:\mathsf{OM}\rightarrow\mathsf{Out}.\,\forall z:\mathsf{OM}.\\ \quad \{P\}\,E1\,\{\mathsf{expr}(\lambda x:\mathsf{OM}.\,\lambda v:\mathsf{Out}.\,R\,x\;\wedge v=\mathit{expr}\,x)\}\\ \quad \{\lambda x:\mathsf{OM}.\,R\,x\;\wedge\;x=z\}\,E2\,\{\mathsf{expr}(\lambda x:\mathsf{OM}.\,\lambda v:\mathsf{Out}.\,Q\,x\;(\mathit{expr}\,z\oplus v))\}\end{array}}{\{P\}\,E1\oplus E2\,\{\mathsf{expr}(Q)\}}$$

Total binary operator $\oplus$: Out $\rightarrow$ Out $\rightarrow$ Out2 rule:

$$\frac{\begin{array}{l}\exists \mathit{expr}:\mathsf{OM}\rightarrow\mathsf{Out}.\,\forall z:\mathsf{OM}.\\ \quad [P]\,E1\,[\mathsf{expr}(\lambda x:\mathsf{OM}.\,\lambda v:\mathsf{Out}.\,R\,x\;\wedge v=\mathit{expr}\,x)]\\ \quad [\lambda x:\mathsf{OM}.\,R\,x\;\wedge\;x=z]\,E2\,[\mathsf{expr}(\lambda x:\mathsf{OM}.\,\lambda v:\mathsf{Out}.\,Q\,x\;(\mathit{expr}\,z\oplus v))]\end{array}}{[P]\,E1\oplus E2\,[\mathsf{expr}(Q)]}$$

Partial ref_assign_at rule:

$$\frac{\begin{array}{l}\exists \mathit{refpos}:\mathsf{OM}\rightarrow\mathsf{MemLoc}.\,\exists \mathit{index}:\mathsf{OM}\rightarrow\mathsf{int}.\,\forall z:\mathsf{OM}.\,\forall w:\mathsf{OM}.\\ \quad \{P\}\\ \quad \mathit{array\_expr}\\ \quad \{\mathsf{expr}(\lambda x:\mathsf{Self}.\,\lambda v:\mathsf{RefType}.\;\; R\,x\;\wedge\\ \qquad\qquad\qquad\qquad \mathsf{CASE}\;v\;\mathsf{OF}\;\{\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{null}\mapsto\mathsf{true}\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{ref}\,r\mapsto r=\mathit{refpos}\,x\})\}\\ \quad \{\lambda x:\mathsf{OM}.\,R\,x\;\wedge\;x=z\}\\ \quad\; \mathit{index\_expr}\\ \quad \{\mathsf{expr}(\lambda x:\mathsf{Self}.\,\lambda v:\mathsf{int}.\,S\,x\;\wedge\;v=\mathit{index}\;\wedge\;\mathit{refpos}\,x=\mathit{refpos}\,z)\}\\ \quad \{\lambda x:\mathsf{OM}.\,S\,x\;\wedge\;x=w\}\\ \quad\; \mathit{data\_expr}\\ \quad \{\mathsf{expr}(\lambda x:\mathsf{Self}.\,\lambda v:\mathsf{RefType}.\,Q(\mathsf{put\_ref}(\mathsf{heap}(\;\mathsf{ml}=\mathit{refpos}\,w,\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{cl}=\mathit{index}\,w\;))\,x(v))(v))\}\end{array}}{\{P\}\,\mathsf{ref\_assign\_at}(\mathit{array\_expr},\mathit{index\_expr})(\mathit{data\_expr})\,\{\mathsf{expr}(Q)\}}$$

Total ref_assign_at rule:

$$
\begin{array}{l}
\exists \mathit{refpos}\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \exists \mathit{index}\colon \mathsf{OM} \to \mathsf{int}.\ \forall z\colon \mathsf{OM}.\ \forall w\colon \mathsf{OM}. \\
\quad [P] \\
\quad \mathit{array\_expr} \\
\quad [\mathsf{expr}(\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{false} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{ref}\, r \mapsto r = \mathit{refpos}\, x\})] \\
\quad [\lambda x\colon \mathsf{OM}.\ R\,x \ \wedge\ x = z] \\
\quad \mathit{index\_expr} \\
\quad [\mathsf{expr}(\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{int}.\ \ S\,x \ \wedge\ v = \mathit{index} \wedge \\
\qquad\qquad\qquad\qquad\qquad\quad 0 \le v \ \wedge\ v < (\mathsf{get\_dimlen}\ (\mathit{refpos}\, z)\ z) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{refpos}\, x = \mathit{refpos}\, z)] \\
\quad [\lambda x\colon \mathsf{OM}.\ S\,x \ \wedge\ x = w] \\
\quad \mathit{data\_expr} \\
\quad [\mathsf{expr}(\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{RefType}.\ \ \mathsf{CASE}\ v\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{true} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{ref}\, r \mapsto \mathsf{SubClass?}\ \ (\mathsf{get\_type}\, r\ x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{get\_type}\ (\mathit{refpos}\, z)\ z)\ \} \wedge \\
\qquad\qquad\qquad\qquad\qquad\quad Q(\mathsf{put\_ref}(\mathsf{heap}(\ \ \mathsf{ml} = \mathit{refpos}\, w, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{cl} = \mathit{index}\, w\ ))\, x\,(v))(v))] \\
\hline
\quad [P]\ \mathsf{ref\_assign\_at}(\mathit{array\_expr}, \mathit{index\_expr})(\mathit{data\_expr})\ [\mathsf{expr}(Q)]
\end{array}
$$

Partial prim_assign_at rule:

$$
\begin{array}{l}
\exists \mathit{refpos}\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \exists \mathit{index}\colon \mathsf{OM} \to \mathsf{int}.\ \forall z\colon \mathsf{OM}.\ \forall w\colon \mathsf{OM}. \\
\quad \{P\} \\
\quad \mathit{array\_expr} \\
\quad \{\mathsf{expr}(\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{true} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{ref}\, r \mapsto r = \mathit{refpos}\, x\})\} \\
\quad \{\mathsf{expr}(\lambda x\colon \mathsf{OM}.\ R\,x \ \wedge\ x = z\} \\
\quad \mathit{index\_expr} \\
\quad \{\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{int}.\ S\,x \ \wedge\ v = \mathit{index} \ \wedge\ \mathit{refpos}\, x = \mathit{refpos}\, z)\} \\
\quad \{\lambda x\colon \mathsf{OM}.\ S\,x \ \wedge\ x = w\} \\
\quad \mathit{data\_expr} \\
\quad \{\mathsf{expr}(\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{RefType}.\ Q(\mathsf{put\_type}(\mathsf{heap}(\ \ \mathsf{ml} = \mathit{refpos}\, w, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{cl} = \mathit{index}\, w\ ))\, x\,(v))(v))\} \\
\hline
\quad \{P\}\ \mathsf{prim\_assign\_at}(\mathsf{put\_type}, \mathit{array\_expr}, \mathit{index\_expr})(\mathit{data\_expr})\ \{\mathsf{expr}(Q)\}
\end{array}
$$

Total prim_assign_at rule:

$$\frac{\begin{array}{l}\exists refpos\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \exists index\colon \mathsf{OM} \to \mathsf{int}.\ \forall z\colon \mathsf{OM}.\ \forall w\colon \mathsf{OM}.\\ \quad [P]\\ \quad array\_expr\\ \quad [\mathsf{expr}(\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge\\ \qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{\\ \qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{false}\\ \qquad\qquad\quad \mid \mathsf{ref}\, r \mapsto r = refpos\, x\})]\\ \quad [\lambda x\colon \mathsf{OM}.\ R\,x\ \wedge\ x = z]\\ \quad index\_expr\\ \quad [\mathsf{expr}(\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{int}.\ \ S\,x\ \wedge\ v = index \wedge\\ \qquad\qquad 0 \le v\ \wedge\ v < (\mathsf{get\_dimlen}\ (refpos\, z)\ z)\\ \qquad\qquad refpos\, x = refpos\, z)]\\ \quad [\lambda x\colon \mathsf{OM}.\ S\,x\ \wedge\ x = w]\\ \quad data\_expr\\ \quad [\mathsf{expr}(\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{RefType}.\ \ Q(\mathsf{put\_type}(\mathsf{heap}(\ \mathsf{ml} = refpos\, w,\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{cl} = index\, w\ ))\, x\, (v))(v))]\end{array}}{[P]\ \mathsf{prim\_assign\_at}(\mathsf{put\_type}, array\_expr, index\_expr)(data\_expr)\ [\mathsf{expr}(Q)]}$$

Partial access_at rule:

$$\frac{\begin{array}{l}\exists refpos\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \forall z\colon \mathsf{OM}.\\ \quad \{P\}\\ \quad array\_expr\\ \quad \{\mathsf{expr}(\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge\\ \qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{\\ \qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{true}\\ \qquad\qquad\quad \mid \mathsf{ref}\, r \mapsto r = refpos\, x\})\}\\ \quad \{\lambda x\colon \mathsf{OM}.\ R\,x\ \wedge\ x = z\}\\ \quad index\_expr\\ \quad \{\mathsf{expr}(\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{RefType}.\ Q\,x\ (\mathsf{get\_type}(\mathsf{heap}(\ \mathsf{ml} = refpos\, w,\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{cl} = v))\, x))\}\end{array}}{\{P\}\ \mathsf{access\_at}(\mathsf{get\_type}, array\_expr, index\_expr)(data\_expr)\ \{\mathsf{expr}(Q)\}}$$

Total access_at rule:

$$\frac{\begin{array}{l}\exists \mathit{refpos}\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \forall z\colon \mathsf{OM}.\\ \quad [P]\\ \quad \mathit{array\_expr}\\ \quad [\mathsf{expr}(\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge\\ \qquad\qquad\qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{false}\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{ref}\, r \mapsto r = \mathit{refpos}\, x\})]\\ \quad [\lambda x\colon \mathsf{OM}.\ R\,x\ \wedge\ x = z]\\ \quad \mathit{index\_expr}\\ \quad [\mathsf{expr}(\lambda x\colon \mathsf{Self}.\ \lambda v\colon \mathsf{int}.\ Q\,x\ (\mathsf{get\_type}(\mathsf{heap}(\ \mathsf{ml} = \mathit{refpos}\, w,\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{cl} = \mathit{index}\, w\ ))\, x))]\end{array}}{[P]\ \mathsf{access\_at}(\mathsf{get\_type}, \mathit{array\_expr}, \mathit{index\_expr})(\mathit{data\_expr})\ [\mathsf{expr}(Q)]}$$

Partial CE2E rule:

$$\frac{\begin{array}{l}\exists \mathit{refpos}\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \exists \mathit{name}\colon \mathsf{OM} \to \mathsf{string}.\ \forall z\colon \mathsf{OM}.\\ \quad \{P\}\\ \quad \mathit{ref\_expr}\\ \quad \{\lambda x\colon \mathsf{OM}.\ \mathsf{expr}(\lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge\\ \qquad\qquad\qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{true}\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{ref}\, r \mapsto\ \ r = \mathit{refpos}\, x \wedge\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{get\_type}\, r\ x = \mathit{name}\, x\})\}\\ \quad \{\lambda x\colon \mathsf{OM}.\ R\,x\ \wedge\ x = z\}\ \mathit{expression}(\mathit{coalg}(\mathit{name}\, z)(\mathit{refpos}\, z))\ \{\mathsf{expr}(Q)\}\end{array}}{\{P\}\ \mathsf{CE2E}(\mathit{coalg})(\mathit{ref\_expr})(\mathit{expression})\ \{\mathsf{expr}(Q)\}}$$

Total CE2E rule:

$$\frac{\begin{array}{l}\exists \mathit{refpos}\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \exists \mathit{name}\colon \mathsf{OM} \to \mathsf{string}.\ \forall z\colon \mathsf{OM}.\\ \quad [P]\\ \quad \mathit{ref\_expr}\\ \quad [\mathsf{expr}(\lambda x\colon \mathsf{OM}.\ \lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge\\ \qquad\qquad\qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{false}\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{ref}\, r \mapsto\ \ r = \mathit{refpos}\, x \wedge\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{get\_type}\, r\ x = \mathit{name}\, x\})]\\ \quad [\lambda x\colon \mathsf{OM}.\ R\,x\ \wedge\ x = z]\ \mathit{expression}(\mathit{coalg}(\mathit{name}\, z)(\mathit{refpos}\, z))\ [\mathsf{expr}(Q)]\end{array}}{[P]\ \mathsf{CE2E}(\mathit{coalg})(\mathit{ref\_expr})(\mathit{expression})\ [\mathsf{expr}(Q)]}$$

Partial CF2F rule:

$$\frac{\begin{array}{l}\exists \mathit{refpos}\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \exists \mathit{name}\colon \mathsf{OM} \to \mathsf{string}.\ \forall z\colon \mathsf{OM}.\\ \quad \{P\}\\ \quad \mathit{ref\_expr}\\ \quad \{\lambda x\colon \mathsf{OM}.\ \mathsf{expr}(\lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge\\ \qquad\qquad\qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{true}\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{ref}\, r \mapsto\ \ r = \mathit{refpos}\, x \wedge\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{get\_type}\, r\, x = \mathit{name}\, x\})\}\\ \quad \{\lambda x\colon \mathsf{OM}.\ R\,x \ \wedge\ x = z\}\ \mathsf{F2E}\ (\mathit{var\_field}(\mathit{coalg}(\mathit{name}\, z)(\mathit{refpos}\, z)))\ \{\mathsf{expr}(Q)\}\end{array}}{\{P\}\ \mathsf{CF2F}(\mathit{coalg})(\mathit{ref\_expr})(\mathit{var\_field})\ \{\mathsf{expr}(Q)\}}$$

Total CF2F rule:

$$\frac{\begin{array}{l}\exists \mathit{refpos}\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \exists \mathit{name}\colon \mathsf{OM} \to \mathsf{string}.\ \forall z\colon \mathsf{OM}.\\ \quad [P]\\ \quad \mathit{ref\_expr}\\ \quad [\mathsf{expr}(\lambda x\colon \mathsf{OM}.\ \lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge\\ \qquad\qquad\qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{false}\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{ref}\, r \mapsto\ \ r = \mathit{refpos}\, x \wedge\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{get\_type}\, r\, x = \mathit{name}\, x\})]\\ \quad [\lambda x\colon \mathsf{OM}.\ R\,x \ \wedge\ x = z]\ \mathsf{F2E}(\mathit{var\_field}(\mathit{coalg}(\mathit{name}\, z)(\mathit{refpos}\, z)))\ [\mathsf{expr}(Q)]\end{array}}{[P]\ \mathsf{CF2F}(\mathit{coalg})(\mathit{ref\_expr})(\mathit{var\_field})\ [\mathsf{expr}(Q)]}$$

Partial CA2A rule:

$$\frac{\begin{array}{l}\exists \mathit{refpos}\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \exists \mathit{name}\colon \mathsf{OM} \to \mathsf{string}.\ \forall z\colon \mathsf{OM}.\\ \quad \{P\}\\ \quad \mathit{ref\_expr}\\ \quad \{\lambda x\colon \mathsf{OM}.\ \mathsf{expr}(\lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge\\ \qquad\qquad\qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{true}\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{ref}\, r \mapsto\ \ r = \mathit{refpos}\, x \wedge\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{get\_type}\, r\, x = \mathit{name}\, x\})\}\\ \quad \{\lambda x\colon \mathsf{OM}.\ R\,x \ \wedge\ x = z\}\ \mathsf{A2E}\ (\mathit{var\_becomes}(\mathit{coalg}(\mathit{name}\, z)(\mathit{refpos}\, z)))\ \mathit{expr}\ \{\mathsf{expr}(Q)\}\end{array}}{\{P\}\ \mathsf{CA2A}(\mathit{coalg})(\mathit{ref\_expr})(\mathit{var\_becomes})(\mathit{expr})\ \{\mathsf{expr}(Q)\}}$$

Total CA2A rule:

$$\frac{\begin{array}{l}\exists \mathit{refpos}\colon \mathsf{OM} \to \mathsf{MemLoc}.\ \exists \mathit{name}\colon \mathsf{OM} \to \mathsf{string}.\ \forall z\colon \mathsf{OM}.\\ \quad [P]\\ \quad \mathit{ref\_expr}\\ \quad [\mathsf{expr}(\lambda x\colon \mathsf{OM}.\ \lambda v\colon \mathsf{RefType}.\ \ R\,x \wedge\\ \qquad\qquad \mathsf{CASE}\ v\ \mathsf{OF}\ \{\\ \qquad\qquad\quad \mid \mathsf{null} \mapsto \mathsf{false}\\ \qquad\qquad\quad \mid \mathsf{ref}\,r \mapsto\ \ r = \mathit{refpos}\,x \wedge\\ \qquad\qquad\qquad\qquad \mathsf{get\_type}\,r\ x = \mathit{name}\,x\})]\\ \quad [\lambda x\colon \mathsf{OM}.\ R\,x\ \wedge\ x = z]\,\mathsf{A2E}(\mathit{var\_becomes}(\mathit{coalg}(\mathit{name}\,z)(\mathit{refpos}\,z)))\ \mathit{expr}\ [\mathsf{expr}(Q)]\end{array}}{[P]\,\mathsf{CA2A}(\mathit{coalg})(\mathit{ref\_expr})(\mathit{var\_becomes})(\mathit{expr})\ [\mathsf{expr}(Q)]}$$

A.3 Exception correctness of statements

$$\begin{array}{l}\mathsf{pre}\colon \mathsf{Self} \to \mathsf{bool},\ \mathsf{post}\colon \mathsf{Self} \to \mathsf{RefType} \to \mathsf{bool},\\ \mathsf{stat}\colon \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}],\ \mathit{str}\colon \mathsf{string} \qquad\qquad \vdash\\ \quad \mathsf{PartialException?}(\mathsf{pre}, \mathsf{stat}, \mathsf{post}, \mathit{str})\ :\ \mathsf{bool} \stackrel{\mathrm{def}}{=}\\ \qquad \forall x\colon \mathsf{Self}.\ \mathsf{pre}\,x \supset\\ \qquad\quad \mathsf{CASE}\ \mathsf{stat}\,x\ \mathsf{OF}\ \{\\ \qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{true}\\ \qquad\qquad \mid \mathsf{norm}\,y \mapsto \mathsf{true}\\ \qquad\qquad \mid \mathsf{abnorm}\,a \mapsto\\ \qquad\qquad\quad \mathsf{CASE}\ a\ \mathsf{OF}\ \{\\ \qquad\qquad\qquad \mid \mathsf{excp}\,e \mapsto\ \ \mathsf{post}\ (e.\mathsf{es})\ (e.\mathsf{ex}) \wedge\\ \qquad\qquad\qquad\qquad\quad \mathsf{CASE}\ e\ \mathsf{OF}\ \{\\ \qquad\qquad\qquad\qquad\qquad \mid \mathsf{null} \mapsto \mathsf{false}\\ \qquad\qquad\qquad\qquad\qquad \mid \mathsf{ref}\,p \mapsto \mathsf{SubClass?}\ (\mathsf{get\_type}\,p\ (e.\mathsf{es}))\ \mathit{str}\ \}\\ \qquad\qquad\qquad \mid \mathsf{rtrn}\,r \mapsto \mathsf{true}\\ \qquad\qquad\qquad \mid \mathsf{break}\,r \mapsto \mathsf{true}\\ \qquad\qquad\qquad \mid \mathsf{cont}\,r \mapsto \mathsf{true}\ \}\}\end{array}$$

$$
\begin{array}{l}
\mathsf{pre}: \mathsf{Self} \to \mathsf{bool}, \mathsf{post}: \mathsf{Self} \to \mathsf{RefType} \to \mathsf{bool}, \\
\mathsf{stat}: \mathsf{Self} \to \mathsf{StatResult[Self]}, \mathit{str}: \mathsf{string} \qquad\qquad \vdash \\
\quad \mathsf{TotalException?(pre, stat, post, \mathit{str})} \ : \ \mathsf{bool} \stackrel{\mathrm{def}}{=} \\
\quad\quad \forall x : \mathsf{Self}.\ \mathsf{pre}\, x \supset \\
\quad\quad\quad \mathsf{CASE\ stat}\, x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{false} \\
\quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto \mathsf{false} \\
\quad\quad\quad\quad \mid \mathsf{abnorm}\, a \mapsto \\
\quad\quad\quad\quad\quad \mathsf{CASE}\ a\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{excp}\, e \mapsto \ \mathsf{post}\ (e.\mathsf{es})\ (e.\mathsf{ex}) \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{CASE}\ e\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{null} \mapsto \mathsf{false} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{ref}\, p \mapsto \mathsf{SubClass?}\ (\mathsf{get\_type}\, p\ (e.\mathsf{es}))\ \mathit{str}\ \} \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{rtrn}\, r \mapsto \mathsf{false} \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{break}\, r \mapsto \mathsf{false} \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{cont}\, r \mapsto \mathsf{false}\ \}\}
\end{array}
$$

Notation:

$$\mathsf{PartialException?}(P, S, Q, \mathit{str}) \stackrel{\mathrm{def}}{=} \{P\}\, S\, \{\mathsf{exception}(Q, \mathit{str})\}$$

$$\mathsf{TotalException?}(P, S, Q, \mathit{str}) \stackrel{\mathrm{def}}{=} [P]\, S\, [\mathsf{exception}(Q, \mathit{str})]$$

Partial exception precondition strengthening:

$$\frac{\forall x : \mathsf{OM}.\ P\, x \supset R\, x \qquad \{R\}\, S\, \{\mathsf{exception}(Q, \mathit{str})\}}{\{P\}\, S\, \{\mathsf{exception}(Q, \mathit{str})\}}$$

Total exception precondition strengthening:

$$\frac{\forall x : \mathsf{OM}.\ P\, x \supset R\, x \qquad [R]\, S\, [\mathsf{exception}(Q, \mathit{str})]}{[P]\, S\, [\mathsf{exception}(Q, \mathit{str})]}$$

Partial exception postcondition weakening:

$$\frac{\forall x : \mathsf{OM}.\ \forall \mathit{str} : \mathsf{string}.\ R\, x\, \mathit{str} \supset Q\, x\, \mathit{str} \qquad \{P\}\, S\, \{\mathsf{exception}(R, \mathit{str})\}}{\{P\}\, S\, \{\mathsf{exception}(Q, \mathit{str})\}}$$

Total exception postcondition weakening:

$$\frac{\forall x : \mathsf{OM}.\ \forall \mathit{str} : \mathsf{string}.\ R\, x\, \mathit{str} \supset Q\, x\, \mathit{str} \qquad [P]\, S\, [\mathsf{exception}(R, \mathit{str})]}{[P]\, S\, [\mathsf{exception}(Q, \mathit{str})]}$$

Partial exception composition rule:

$$\frac{\{P\}\, S\, \{R\} \qquad \{P\}\, S\, \{\mathsf{exception}(Q, \mathit{str})\} \qquad \{R\}\, T\, \{\mathsf{exception}(Q, \mathit{str})\}}{\{P\}\, S\, ;\, T\, \{\mathsf{exception}(Q, \mathit{str})\}}$$

Total exception left composition rule:

$$\frac{[P]\,S\,[\mathsf{exception}(Q, str)]}{[P]\,S\,;\,T\,[\mathsf{exception}(Q, str)]}$$

Total exception right composition rule:

$$\frac{[P]\,S\,[R] \qquad [R]\,T\,[\mathsf{exception}(Q, str)]}{[P]\,S\,;\,T\,[\mathsf{exception}(Q, str)]}$$

Partial exception IF-THEN rule:

$$\frac{\begin{array}{c}\{P\}\,C\,\{\mathsf{exception}(Q, str)\} \\ \{P\ \wedge\ \mathsf{true}(C)\}\,\mathsf{E2S}(C)\,;\,S\,\{\mathsf{exception}(Q, str)\}\end{array}}{\{P\}\,\mathsf{IF\text{-}THEN}(C)(S)\,\{\mathsf{exception}(Q, str)\}}$$

Total exception IF-THEN condition rule:

$$\frac{[P]\,C\,[\mathsf{exception}(Q, str)]}{[P]\,\mathsf{IF\text{-}THEN}(C)(S)\,[\mathsf{exception}(Q, str)]}$$

Total exception IF-THEN rule:

$$\frac{\begin{array}{c}[P]\,C\,[\lambda x : \mathsf{OM}.\,\lambda v : \mathsf{bool}.\,v] \\ [P\ \wedge\ \mathsf{true}(C)]\,\mathsf{E2S}(C)\,;\,S\,[\mathsf{exception}(Q, str)]\end{array}}{[P]\,\mathsf{IF\text{-}THEN}(C)(S)\,[\mathsf{exception}(Q, str)]}$$

Partial exception IF-THEN-ELSE rule:

$$\frac{\begin{array}{c}\{P\}\,C\,\{\mathsf{exception}(Q, str)\} \\ \{P\ \wedge\ \mathsf{true}(C)\}\,\mathsf{E2S}(C)\,;\,S\,\{\mathsf{exception}(Q, str)\} \\ \{P\ \wedge\ \mathsf{false}(C)\}\,\mathsf{E2S}(C)\,;\,T\,\{\mathsf{exception}(Q, str)\}\end{array}}{\{P\}\,\mathsf{IF\text{-}THEN\text{-}ELSE}(C)(S)(T)\,\{\mathsf{exception}(Q, str)\}}$$

Total exception IF-THEN-ELSE condition rule:

$$\frac{[P]\,C\,[\mathsf{exception}(Q, str)]}{[P]\,\mathsf{IF\text{-}THEN\text{-}ELSE}(C)(S)(T)\,[\mathsf{exception}(Q, str)]}$$

Total exception IF-THEN-ELSE rule:

$$\frac{\begin{array}{c}[P]\,C\,[\mathsf{true}] \\ [P\ \wedge\ \mathsf{true}(C)]\,\mathsf{E2S}(C)\,;\,S\,[\mathsf{exception}(Q, str)] \\ [P\ \wedge\ \mathsf{false}(C)]\,\mathsf{E2S}(C)\,;\,T\,[\mathsf{exception}(Q, str)]\end{array}}{[P]\,\mathsf{IF\text{-}THEN\text{-}ELSE}(C)(S)(T)\,[\mathsf{exception}(Q, str)]}$$

Partial exception CATCH-STAT-RETURN rule:

$$\frac{\{P\}\ S\ \{\mathsf{exception}(Q, str)\}}{\{P\}\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(S)\ \{\mathsf{exception}(Q, str)\}}$$

Total exception CATCH-STAT-RETURN rule:

$$\frac{[P]\ S\ [\mathsf{exception}(Q, str)]}{[P]\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(S)\ [\mathsf{exception}(Q, str)]}$$

Partial exception CATCH-BREAK rule:

$$\frac{\{P\}\ S\ \{\mathsf{exception}(Q, str)\}}{\{P\}\ \mathsf{CATCH\text{-}BREAK}(ll)(S)\ \{\mathsf{exception}(Q, str)\}}$$

Total exception CATCH-BREAK rule:

$$\frac{[P]\ S\ [\mathsf{exception}(Q, str)]}{[P]\ \mathsf{CATCH\text{-}BREAK}(ll)(S)\ [\mathsf{exception}(Q, str)]}$$

Partial exception CATCH-CONTINUE rule:

$$\frac{\{P\}\ S\ \{\mathsf{exception}(Q, str)\}}{\{P\}\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ \{\mathsf{exception}(Q, str)\}}$$

Total exception CATCH-CONTINUE rule:

$$\frac{[P]\ S\ [\mathsf{exception}(Q, str)]}{[P]\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ [\mathsf{exception}(Q, str)]}$$

Partial exception WHILE rule:

$$\frac{\begin{array}{c}\{P\ \wedge\ \mathsf{true}(C)\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ \{P\} \\ \{P\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ \{\mathsf{exception}(Q, str)\}\end{array}}{\{P\}\ \mathsf{WHILE}(ll)(C)(S)\ \{\mathsf{exception}(Q, str)\}}$$

Total exception WHILE rule:

$$\frac{\begin{array}{c}\mathsf{well\_founded?}(R) \\ [P]\ \mathsf{TRY\text{-}CATCH}(\mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S))[(str, \lambda r : \mathsf{RefType.\ skip})]\ [\mathsf{true}] \\ \forall a.\{P \wedge \mathsf{true}(C) \wedge variant = a\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ \{P \wedge (variant, a) \in R\} \\ \{P\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ \{\mathsf{exception}(Q, str)\} \\ \{P\ \wedge\ \mathsf{false}(C)\}\ \mathsf{E2S}(C)\ \{\mathsf{false}\}\end{array}}{[P]\ \mathsf{WHILE}(ll)(C)(S)\ [\mathsf{exception}(Q, str)]}$$

Partial exception FOR rule:

$$\frac{\begin{array}{c}\{P \ \wedge \ \mathsf{true}(C)\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ ;\ U\ \{P\} \\ \{P\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ ;\ U\ \{\mathsf{exception}(Q, str)\}\end{array}}{\{P\}\ \mathsf{FOR}(ll)(C)(U)(S)\ \{\mathsf{exception}(Q, str)\}}$$

Total exception FOR rule:

$$\frac{\begin{array}{c}\mathsf{well\_founded?}(R) \\ [P]\ \mathsf{TRY\text{-}CATCH}(\mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ ;\ U)[(str, \lambda r : \mathsf{RefType}.\ \mathsf{skip})]\ [\mathsf{true}] \\ \forall a.\ \{P \ \wedge \ \mathsf{true}(C) \ \wedge \ variant = a\} \\ \quad \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ ;\ U \\ \quad \{P \ \wedge (variant, a) \in R\} \\ \{P\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ ;\ U\ \{\mathsf{exception}(Q, str)\} \\ \{P \ \wedge \ \mathsf{false}(C)\}\ \mathsf{E2S}(C)\ \{\mathsf{false}\}\end{array}}{[P]\ \mathsf{FOR}(ll)(C)(U)(S)\ [\mathsf{exception}(Q, str)]}$$

A.4 Exception correctness of expressions

```
pre: Self → bool, post: Self → RefType → bool,
expr: Self → ExprResult[Self, Out], str: string        ⊢

    PartialException?(pre, expr, post, str) : bool  def=

        ∀x : Self. pre x ⊃
            CASE expr x OF {
              | hang ↦ true
              | norm y ↦ true
              | abnorm a ↦
                    post (e.es) (e.ex) ∧
                    CASE e OF {
                      | null ↦ false
                      | ref p ↦ SubClass? (get_type p (e.es)) str }}
pre: Self → bool, post: Self → RefType → bool,
expr: Self → ExprResult[Self, Out], str: string        ⊢

    TotalException?(pre, expr, post, str) : bool  def=

        ∀x : Self. pre x ⊃
            CASE expr x OF {
              | hang ↦ false
              | norm y ↦ false
              | abnorm a ↦
                    post (e.es) (e.ex) ∧
                    CASE e OF {
                      | null ↦ false
                      | ref p ↦ SubClass? (get_type p (e.es)) str }}
```

Notation:

$$\text{PartialException?}(P, E, Q, str) \stackrel{\text{def}}{=} [P]\, E\, [\text{exception}(Q, str)]$$

$$\text{TotalNormal?}(P, E, Q, str) \stackrel{\text{def}}{=} [P]\, E\, [\text{exception}(Q, str)]$$

Partial exception E2S rule:

$$\frac{\{P\}\, E\, \{\text{exception}(\lambda x : \text{OM}.\, \lambda v : \text{Out}.\, Q\, x, str)\}}{\{P\}\, \text{E2S}(E)\, \{\text{exception}(Q, str)\}}$$

Total exception E2S rule:

$$\frac{[P]\, E\, [\text{exception}(\lambda x : \text{OM}.\, \lambda v : \text{Out}.\, Q\, x, str)]}{[P]\, \text{E2S}(E)\, [\text{exception}(Q, str)]}$$

Partial exception expression precondition strengthening:

$$\frac{\forall x : \text{OM}.\, P\, x \supset R\, x \qquad \{R\}\, E\, \{\text{exception}(Q, str)\}}{\{P\}\, E\, \{\text{exception}(Q, str)\}}$$

Total exception expression precondition strengthening:

$$\frac{\forall x : \text{OM}.\, P\, x \supset R\, x \qquad [R]\, E\, [\text{exception}(Q, str)]}{[P]\, E\, [\text{exception}(Q, str)]}$$

Partial exception expression postcondition weakening:

$$\frac{\forall x : \text{OM}.\, \forall str : \text{string}.\, R\, x\, str \supset Q\, x\, str \qquad \{P\}\, E\, \{\text{exception}(R, str)\}}{\{P\}\, E\, \{\text{exception}(Q, str)\}}$$

Total exception expression postcondition weakening:

$$\frac{\forall x : \text{OM}.\, \forall str : \text{string}.\, R\, x\, str \supset Q\, x\, str \qquad [P]\, E\, [\text{exception}(R, str)]}{[P]\, E\, [\text{exception}(Q, str)]}$$

Partial exception assignment rule:

$$\frac{\{P\}\, E\, \{\text{exception}(Q, str)\}}{\{P\}\, \text{A2E}(var\_becomes)(E)\, \{\text{exception}(Q, str)\}}$$

Total exception assignment rule:

$$\frac{[P]\, E\, [\text{exception}(Q, str)]}{[P]\, \text{A2E}(var\_becomes)(E)\, [\text{exception}(Q, str)]}$$

Partial exception CATCH-EXPR-RETURN rule:

$$\frac{\{P\}\, S\, \{\text{exception}(Q, str)\}}{\{P\}\, \text{CATCH-EXPR-RETURN}(S)\, \{\text{exception}(Q, str)\}}$$

Total exception CATCH-EXPR-RETURN rule:

$$\frac{[P]\, S\, [\text{exception}(Q, str)]}{[P]\, \text{CATCH-EXPR-RETURN}(S)\, [\text{exception}(Q, str)]}$$

A.5 Return correctness of statements

$$
\begin{array}{l}
\text{pre, post}: \text{Self} \to \text{bool}, \text{stat}: \text{Self} \to \text{StatResult[Self]} \vdash \\
\quad \text{PartialReturn?(pre, stat, post)} : \text{bool} \stackrel{\text{def}}{=} \\
\quad\quad \forall x : \text{Self}.\, \text{pre}\, x \supset \text{CASE stat}\, x \text{ OF } \{ \\
\quad\quad\quad\quad \mid \text{hang} \mapsto \text{true} \\
\quad\quad\quad\quad \mid \text{norm}\, y \mapsto \text{true} \\
\quad\quad\quad\quad \mid \text{abnorm}\, a \mapsto \text{CASE}\, a \text{ OF } \{ \\
\quad\quad\quad\quad\quad\quad \mid \text{excp}\, e \mapsto \text{true} \\
\quad\quad\quad\quad\quad\quad \mid \text{rtrn}\, r \mapsto \text{post}\, r \\
\quad\quad\quad\quad\quad\quad \mid \text{break}\, r \mapsto \text{true} \\
\quad\quad\quad\quad\quad\quad \mid \text{cont}\, r \mapsto \text{true} \}\}
\end{array}
$$

$$
\begin{array}{l}
\text{pre, post}: \text{Self} \to \text{bool}, \text{stat}: \text{Self} \to \text{StatResult[Self]} \vdash \\
\quad \text{TotalReturn?(pre, stat, post)} : \text{bool} \stackrel{\text{def}}{=} \\
\quad\quad \forall x : \text{Self}.\, \text{pre}\, x \supset \text{CASE stat}\, x \text{ OF } \{ \\
\quad\quad\quad\quad \mid \text{hang} \mapsto \text{false} \\
\quad\quad\quad\quad \mid \text{norm}\, y \mapsto \text{false} \\
\quad\quad\quad\quad \mid \text{abnorm}\, a \mapsto \text{CASE}\, a \text{ OF } \{ \\
\quad\quad\quad\quad\quad\quad \mid \text{excp}\, e \mapsto \text{false} \\
\quad\quad\quad\quad\quad\quad \mid \text{rtrn}\, r \mapsto \text{post}\, r \\
\quad\quad\quad\quad\quad\quad \mid \text{break}\, r \mapsto \text{false} \\
\quad\quad\quad\quad\quad\quad \mid \text{cont}\, r \mapsto \text{false} \}\}
\end{array}
$$

Notation:

$$\text{PartialReturn?}(P, S, Q) \stackrel{\text{def}}{=} \{P\}\, S\, \{\text{return}(Q)\}$$

$$\text{TotalReturn?}(P, S, Q) \stackrel{\text{def}}{=} [P]\, S\, [\text{return}(Q)]$$

Partial return precondition strengthening:

$$\frac{\forall x : \text{OM}.\, P\, x \supset R\, x \qquad \{R\}\, S\, \{\text{return}(Q)\}}{\{P\}\, S\, \{\text{return}(Q)\}}$$

Total return precondition strengthening:

$$\frac{\forall x : \text{OM}.\, P\, x \supset R\, x \qquad [R]\, S\, [\text{return}(Q)]}{[P]\, S\, [\text{return}(Q)]}$$

Partial return postcondition weakening:

$$\frac{\forall x : \text{OM}.\, R\, x \supset Q\, x \qquad \{P\}\, S\, \{\text{return}(R)\}}{\{P\}\, S\, \{\text{return}(Q)\}}$$

Total return postcondition weakening:

$$\frac{\forall x : \text{OM}.\, R\, x \supset Q\, x \qquad [P]\, S\, [\text{return}(R)]}{[P]\, S\, [\text{return}(Q)]}$$

Partial return composition rule:

$$\frac{\{P\}\, S\, \{R\} \qquad \{P\}\, S\, \{\mathsf{return}(Q)\} \qquad \{R\}\, T\, \{\mathsf{return}(Q)\}}{\{P\}\, S\,;\, T\, \{\mathsf{return}(Q)\}}$$

Total return left composition rule:

$$\frac{[P]\, S\, [\mathsf{return}(Q)]}{[P]\, S\,;\, T\, [\mathsf{return}(Q)]}$$

Total return right composition rule:

$$\frac{[P]\, S\, [R] \qquad [R]\, T\, [\mathsf{return}(Q)]}{[P]\, S\,;\, T\, [\mathsf{return}(Q)]}$$

Partial return IF-THEN rule:

$$\frac{\begin{array}{c}\{P \ \wedge\ \mathsf{true}(C)\}\ \mathsf{E2S}(C)\,;\, S\, \{\mathsf{return}(Q)\} \\ \{P \ \wedge\ \mathsf{false}(C)\}\ \mathsf{E2S}(C)\, \{\mathsf{return}(Q)\}\end{array}}{\{P\}\ \mathsf{IF\text{-}THEN}(C)(S)\, \{\mathsf{return}(Q)\}}$$

Total return IF-THEN rule:

$$\frac{\begin{array}{c}[P \ \wedge\ \mathsf{true}(C)]\ \mathsf{E2S}(C)\,;\, S\, [\mathsf{return}(Q)] \\ [P \ \wedge\ \mathsf{false}(C)]\ \mathsf{E2S}(C)\, [\mathsf{return}(Q)]\end{array}}{[P \ \wedge\ \mathsf{norm}(C)]\ \mathsf{IF\text{-}THEN}(C)(S)\, [\mathsf{return}(Q)]}$$

Partial return IF-THEN-ELSE rule:

$$\frac{\begin{array}{c}\{P \ \wedge\ \mathsf{true}(C)\}\ \mathsf{E2S}(C)\,;\, S\, \{\mathsf{return}(Q)\} \\ \{P \ \wedge\ \mathsf{false}(C)\}\ \mathsf{E2S}(C)\,;\, T\, \{\mathsf{return}(Q)\}\end{array}}{\{P\}\ \mathsf{IF\text{-}THEN\text{-}ELSE}(C)(S)(T)\, \{\mathsf{return}(Q)\}}$$

Total return IF-THEN-ELSE rule:

$$\frac{\begin{array}{c}[P \ \wedge\ \mathsf{true}(C)]\ \mathsf{E2S}(C)\,;\, S\, [\mathsf{return}(Q)] \\ [P \ \wedge\ \mathsf{false}(C)]\ \mathsf{E2S}(C)\,;\, T\, [\mathsf{return}(Q)]\end{array}}{[P \ \wedge\ \mathsf{norm}(C)]\ \mathsf{IF\text{-}THEN\text{-}ELSE}(C)(S)(T)\, [\mathsf{return}(Q)]}$$

Partial RETURN axiom:

$$\{P\}\ \mathsf{RETURN}\, \{\mathsf{return}(P)\}$$

Total RETURN axiom:

$$[P]\ \mathsf{RETURN}\, [\mathsf{return}(P)]$$

Partial return CATCH-STAT-RETURN rule:

$$\frac{\{P\}\ S\ \{\mathsf{return}(Q)\} \qquad \{P\}\ S\ \{Q\}}{\{P\}\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(S)\ \{Q\}}$$

Total return CATCH-STAT-RETURN return rule:

$$\frac{[P]\ S\ [\mathsf{return}(Q)]}{[P]\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(S)\ [Q]}$$

Partial return CATCH-EXPR-RETURN rule:

$$\frac{\{P\}\ S\ \{\mathsf{return}(\lambda x:\ Q\ x\ (v\ x).)\}}{\{P\}\ \mathsf{CATCH\text{-}EXPR\text{-}RETURN}(S)(v)\ \{Q\}}$$

Total return CATCH-EXPR-RETURN rule:

$$\frac{[P]\ S\ [\mathsf{return}(\lambda x:\ Q\ x\ (v\ x).)]}{[P]\ \mathsf{CATCH\text{-}EXPR\text{-}RETURN}(S)(v)\ [Q]}$$

Partial return CATCH-BREAK rule:

$$\frac{\{P\}\ S\ \{\mathsf{return}(Q)\}}{\{P\}\ \mathsf{CATCH\text{-}BREAK}(ll)(S)\ \{\mathsf{return}(Q)\}}$$

Total return CATCH-BREAK rule:

$$\frac{[P]\ S\ [\mathsf{return}(Q)]}{[P]\ \mathsf{CATCH\text{-}BREAK}(ll)(S)\ [\mathsf{return}(Q)]}$$

Partial return CATCH-CONTINUE rule:

$$\frac{\{P\}\ S\ \{\mathsf{return}(Q)\}}{\{P\}\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ \{\mathsf{return}(Q)\}}$$

Total return CATCH-CONTINUE rule:

$$\frac{[P]\ S\ [\mathsf{return}(Q)]}{[P]\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ [\mathsf{return}(Q)]}$$

Partial return WHILE rule:

$$\frac{\begin{array}{c}\{P\ \wedge\ \mathsf{true}(C)\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ \{P\} \\ \{P\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ \{\mathsf{return}(Q)\}\end{array}}{\{P\}\ \mathsf{WHILE}(ll)(C)(S)\ \{\mathsf{return}(Q)\}}$$

Total return WHILE rule:

$$\frac{\begin{array}{c}\mathsf{well\_founded?}(R) \\ [P]\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(\mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S))\ [\mathsf{true}] \\ \forall a.\ \{P\ \wedge\ \mathsf{true}(C)\ \wedge\ \mathit{variant} = a\} \\ \quad \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S) \\ \quad \{P\ \wedge\ \mathsf{true}(C)\ \wedge\ (\mathit{variant}, a) \in R\} \\ \{P\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ \{\mathsf{return}(Q)\}\end{array}}{[P]\ \mathsf{WHILE}(ll)(C)(S)\ [\mathsf{return}(Q)]}$$

Partial return FOR rule:

$$\frac{\begin{array}{c}\{P\ \wedge\ \mathsf{true}(C)\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ ;\ U\ \{P\} \\ \{P\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ ;\ U\ \{\mathsf{return}(Q)\}\end{array}}{\{P\}\ \mathsf{FOR}(ll)(C)(U)(S)\ \{\mathsf{return}(Q)\}}$$

Total return FOR rule:

$$\frac{\begin{array}{c}\mathsf{well\_founded?}(R) \\ [P]\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(\mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ ;\ U)\ [\mathsf{true}] \\ \forall a.\ \{P\ \wedge\ \mathsf{true}(C)\ \wedge\ \mathit{variant} = a\} \\ \quad \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ ;\ U \\ \quad \{P\ \wedge\ \mathsf{true}(C)\ \wedge\ (\mathit{variant}, a) \in R\} \\ \{P\}\ \mathsf{E2S}(C)\ ;\ \mathsf{CATCH\text{-}CONTINUE}(ll)(S)\ ;\ U\ \{\mathsf{return}(Q)\}\end{array}}{[P]\ \mathsf{FOR}(ll)(C)(U)(S)\ [\mathsf{return}(Q)]}$$

Samenvatting

Programma correctheid is altijd een belangrijk onderzoeksonderwerp geweest binnen de informatica. Idealiter wordt elk programma correct bewezen, dat wil zeggen: er wordt formeel aangetoond dat het programma aan zijn (formele) specificatie voldoet. Al sinds de jaren zestig worden er bewijsmethoden voorgesteld waarmee programma's correct bewezen kunnen worden en theoretisch is bekend hoe correctheidsbewijzen geconstrueerd kunnen worden.

Echter, deze bewijsmethoden beperken zich meestal tot programmeertalen met een eenvoudige semantiek en ze zijn vooral geschikt voor kleine programma's, omdat er in het bewijs veel kleine stapjes gemaakt moeten worden. De in de praktijk gebruikte programmeertalen en programma's zijn daardoor niet direct geschikt voor deze bewijsmethoden: de programmeertalen hebben vaak een ingewikkelde semantiek en de programma's die geverifieerd zouden moeten worden zijn veel groter dan voor een mens te behapstukken is.

Het LOOP project (waarbij LOOP staat voor *Logic of Object Oriented Programming* oftewel de logica van het object-georiënteerd programmeren) richt zich op het gebruik van formele methoden voor object-georiënteerde (programmeer- en specificatie-)talen. Dit proefschrift beschrijft een onderdeel van het LOOP project dat zich speciaal richt op het gebruik van formele methoden en het redeneren over programma's geschreven in de programmeertaal JAVA. JAVA is een volop gebruikte, object-georiënteerde programmeertaal met een onduidelijke semantiek. In dit proefschrift wordt een semantiek gegeven voor het sequentiële gedeelte van deze programmeertaal. Deze semantiek houdt rekening met allerlei 'vieze' details van de taal, zoals *exceptions*, zij-effecten in de evaluatie van expressies, en de mogelijkheid om plotseling uit een while-loop te breken.

Voor het redeneren wordt gebruik gemaakt van zogenaamde stellingbewijzers, dit zijn programma's die de gebruiker ondersteunen bij het bewijzen van een (wiskundige) stelling. De gebruiker geeft aan welke stap hij wil nemen in het bewijs en het systeem voert deze uit. Het voordeel van deze benadering is dat het systeem zorgt dat elke stap correct wordt uitgevoerd en dat het systeem bij houdt welke takken van het bewijs nog open zijn. Naast deze stellingbewijzers wordt gebruik gemaakt van een *compiler*, die JAVA programma's omzet in input voor deze stellingbewijzers. De theorieën die gegenereerd worden door de compiler beschrijven precies de semantiek van de vertaalde klassen.

Hoofdstuk 1 van dit proefschrift beschrijft de achtergrond en plaatst dit proefschrift binnen het kader van het LOOP project. Ook wordt hier een heel beknopte inleiding gegeven op object-oriëntatie.

Hoofdstuk 2 beschrijft de semantiek van sequentieel JAVA. Het eerste gedeelte beschrijft de zogenaamde *semantical prelude*, een verzameling definities die gebruikt kunnen worden als basis om de semantiek van een programma te beschrijven. Deze semantical prelude beschrijft de semantiek van statements en expressies en het geheugenmodel dat gebruikt wordt. Het laatste gedeelte van dit hoofdstuk beschrijft hoe er semantiek gegeven wordt aan een programma door

de klassenstructuur op een bepaalde manier te vertalen.

Hoofdstuk 3 introduceert de twee stellingbewijzers die in het proefschrift gebruikt worden: PVS en ISABELLE. Beide stellingbewijzers worden uitgebreid geïntroduceerd en er wordt uitgelegd hoe de semantical prelude beschreven is in de taal van deze stellingbewijzers. Vervolgens worden beide systemen met elkaar vergeleken, wat een beschrijving oplevert van de ideale stellingbewijzer.

In Hoofdstuk 4 wordt het LOOP tool beschreven. Dit is een compiler die JAVA klassen omzet in een semantische beschrijving, in de specificatietaal van PVS of ISABELLE. Ook worden hier enkele kleine, maar niet-triviale JAVA programma verificaties beschreven.

Hoofdstuk 5 presenteert een speciale Hoare logica voor JAVA. Met behulp van deze logica is het eenvoudiger om over programma's met bijvoorbeeld loops te redeneren. Kenmerkend voor deze Hoare logica is dat deze rekening houdt met zij-effecten en met abrupte terminatie. In het bijzonder worden er regels gepresenteerd waarmee bewezen kan worden dat een loop abrupt termineert, bijvoorbeeld omdat er een *exception* optreedt.

Hoofdstuk 6 beschrijft JML, de *Java Modeling Language*. Dit is een taal waarmee specificaties van JAVA klassen geschreven kunnen worden. De expressies in JML gebruiken JAVA syntax, met enkele uitbreidingen en beperkingen. Op basis van de specificaties kunnen bewijsverplichtingen voor de klassen gegenereerd worden. De generatie van bewijsverplichtingen is lopend onderzoek. Het gebruik van JML leidt ook tot een meer modulaire stijl van bewijzen, waarbij specificaties van klassen (of methoden) gebruikt worden om andere klassen (of methoden) correct te bewijzen. In dit hoofdstuk wordt ook verder ingegaan op een aantal typische aspecten van modulaire verificatie.

In Hoofdstuk 7 worden twee *case studies* gepresenteerd. Beide case studies verifiëren één van de klassen uit JAVA's standaard klassenbibliotheek. De eerste case study is de verificatie van een invariant over de klasse `Vector`: er wordt aangetoond dat een bepaalde integriteitsconstraint (namelijk dat er nooit meer elementen worden opgeslagen in een vector dan er capaciteit is) behouden wordt door alle methoden van de klasse. De tweede case study verifieert een functionele specificatie van de klasse `Collection`, dat wil zeggen dat voor elke methode aangetoond wordt wat het effect is op de gehele collection.

Tenslotte worden er in Hoofdstuk 8 een aantal afsluitende opmerkingen gemaakt en wordt nader ingegaan op de vraag welke stellingbewijzer geschikter is voor het correct bewijzen van JAVA programma's (in onze benadering).

Curriculum Vitae

May 3, 1973	born in Utrecht, Netherlands
August 1985 – May 1991	VWO Montessori Lyceum Herman Jordan, Zeist, Netherlands
September 1, 1991 – August 1996	Student of Computer Science Utrecht University, Netherlands
September 1, 1996 – August 31, 2000	PhD student University of Nijmegen, Netherlands
October 1, 2000 –	Post Doc INRIA Sophia-Antipolis Projet Oasis Sophia-Antipolis, France

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-1

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-2

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-3

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-4

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-5

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-6

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-7

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-8

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-9

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03